

UNIVERSIDADE FEDERAL DE MATO GROSSO COORDENAÇÃO DE ENSINO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

CRIAÇÃO DE UM FRAMEWORK PHP PARA AGILIZAR O PROCESSO DE DESENVOLVIMENTO DE SISTEMAS WEB

**HUDSON FRANCIS VENTURA DE SOUZA**

CUIABÁ – MT

2017

UNIVERSIDADE FEDERAL DE MATO GROSSO

COORDENAÇÃO DE ENSINO DE GRADUAÇÃO EM

SISTEMAS DE INFORMAÇÃO

CRIAÇÃO DE UM FRAMEWORK PHP PARA AGILIZAR O PROCESSO DE DESENVOLVIMENTO DE SISTEMAS WEB

**HUDSON FRANCIS VENTURA DE SOUZA**

Relatório apresentado ao Instituto de Computação da Universidade Federal de Mato Grosso, para obtenção do título de Bacharel em Sistemas de Informação.

CUIABÁ – MT

2017

UNIVERSIDADE FEDERAL DE MATO GROSSO

COORDENAÇÃO DE ENSINO DE GRADUAÇÃO EM

SISTEMAS DE INFORMAÇÃO

HUDSON FRANCIS VENTURA DE SOUZA

Relatório de Estágio Supervisionado apresentado à Coordenação do Curso de Sistemas de Informação como uma das exigências para obtenção do título de Bacharel em Sistemas de Informação da Universidade Federal de Mato Grosso

Aprovado por:

Prof. Nilton Hideki Takagi

Instituto de Computação

(COORDENADOR DE ESTÁGIOS)

Prof. Thiago Meireles Ventura

Instituto de Computação

(ORIENTADOR)

Clariana Souza Cardoso

Amaggi Exportação e Importação LTDA.

(SUPERVISOR TECNOLOGIA DA INFORMAÇÃO)

Prof. Daniel Ávila Vecchiato

Instituto de Computação

(CONVIDADO)

**DEDICATÓRIA**

*Dedicado a todos os professores que tive ao longo de minha vida acadêmica, sem eles, não conseguiria chegar até onde cheguei, e graças a cada um, pude construir e desenvolver uma base para meu conhecimento e paixão pela tecnologia.*

*Em especial quero dedicar este trabalho aos professores; Thiago Meirelles Ventura, Nilton Takagi, Nelcileno Araujo e Jivago Medeiros, que foram além, que conseguiram transmitir um pouco do que vivencio hoje. O conhecimento não foi atrelado somente à sala de aula ou nos trabalhos desenvolvidos, mas, em tudo aquilo que foi vivido, nas ideias trocadas, dúvidas que foram brilhantemente respondidas e que, graças ao respeito mútuo entre aluno-professor, consegui desenvolver um perfil de desenvolvedor apaixonado pela ciência de forma geral e pela tecnologia.*

*Levarei por toda minha vida as experiências e conhecimento que adquiri durante o curso, e as amizades as quais criei.*

*Dedico este trabalho ainda a todos os amantes da ciência e tecnologia, que tentei de forma simples expressar parte do meu conhecimento para que seja público e que possa servir de inspiração para alguém.*

AGRADECIMENTOS

Agradeço a Deus, por me proteger de qualquer obstáculo que pudesse me impedir de atingir meus objetivos. Aos meus pais Francisco Agostinho e Aparecida Ventura, por serem as pessoas que inseriram na vida acadêmica e profissional, mesmo quando eu nem tinha noção do que valia o estudo e me possibilitando chegar onde cheguei. À minha esposa Adriana Pereira dos Santos por cuidar da minha filha Lanna Sophie Scarlett sempre que não me fiz presente, por me auxiliar pontualmente em determinadas atividades do dia-a-dia e compreender com paciência esta etapa de minha vida.

Ao Wagner Biasi, Clariana Cardoso e Henrique Haubert que acreditaram no meu potencial como desenvolvedor e me deram uma chance de mostrar meu potencial. Aos professores por todo conhecimento compartilhado durante o curso, em especial ao meu orientador Thiago Meireles Ventura, que mesmo com seu tempo escasso se empenhou muito em analisar este relatório minuciosamente e orientar sobre os assuntos abordados.

Aos colegas de faculdade por direta e indiretamente colaborar com a absorção de conhecimento, seja acadêmico ou experiência de vida. E por fim a todas as pessoas ainda não citadas que me ajudaram de alguma forma a concluir este curso com êxito.

Sumário

[LISTA DE FIGURAS 8](#_Toc478255976)

[LISTA DE SIGLAS E ABREVIATURAS 10](#_Toc478255977)

[RESUMO 11](#_Toc478255978)

[INTRODUÇÃO 12](#_Toc478255979)

[1. REVISÃO DE LITERATURA 13](#_Toc478255980)

[1.1. LINGUAGENS DE DESENVOLVIMENTO *WEB* 13](#_Toc478255981)

[1.2. O *FRAMEWORK* 14](#_Toc478255982)

[1.3. O MODELO *MVC* 14](#_Toc478255983)

[1.4. *FRAMEWORKS* *MVC* ATUAIS 15](#_Toc478255984)

[2. MATERIAS, TÉCNICAS E MÉTODOS 18](#_Toc478255985)

[2.1.1. WAMPP 18](#_Toc478255986)

[2.1.2. PHPDESIGNER 19](#_Toc478255987)

[2.1.3. SOFTEXPERT 20](#_Toc478255988)

[2.1.4. NAVICAT 21](#_Toc478255989)

[2.1.5. POSTGRESQL 22](#_Toc478255990)

[2.1.6. ORACLE 22](#_Toc478255991)

[3. RESULTADOS 23](#_Toc478255992)

[3.1. INTRODUÇÃO AO FUNCIONAMENTO DO *FRAMEWORK* 23](#_Toc478255993)

[3.2. INDEX.PHP E O INICIO DO PROJETO 24](#_Toc478255994)

[3.3. MODULARIZAÇÃO 25](#_Toc478255995)

[3.4. ENDEREÇOS AMIGÁVEIS E A REESCRITA DE *URL* 27](#_Toc478255996)

[3.4.1. O FUNCIONAMENTO DA REESCRITA DE *URL* 28](#_Toc478255997)

[3.4.2. SISTEMA DE ROTEAMENTO 30](#_Toc478255998)

[3.4.3. DEPENDENCIAS *HTML* 32](#_Toc478255999)

[3.5. INICIANDO A SEGURANÇA DO PROJETO 34](#_Toc478256000)

[3.6. ESTRUTURA DE CLASSES DO *FRAMEWORK* 36](#_Toc478256001)

[3.6.1. A CAMADA CONTROLLER 37](#_Toc478256002)

[3.6.2. A CAMADA VIEW 38](#_Toc478256003)

[3.6.3. CARREGAMENTO DE BIBLIOTECAS 40](#_Toc478256004)

[3.6.4. A CAMADA MODEL 41](#_Toc478256005)

[3.6.5. BILIOTECAS 43](#_Toc478256006)

[3.6.5.1. BIBLIOTECA DE MANIPULAÇÃO DE DADOS 43](#_Toc478256007)

[3.7. ESTADO DO *FRAMEWORK* 47](#_Toc478256008)

[3.8. APLICAÇÃO E USABILIDADE 49](#_Toc478256009)

[3.8.1. O RELATÓRIO DE NÃO CONFORMIDADES EM PLANILHAS 49](#_Toc478256010)

[3.8.1.1. O RELATÓRIO DE NÃO CONFORMIDADES INTERFACE *WEB* 50](#_Toc478256011)

[3.8.2. APLICAÇÃO DO *FRAMEWORK* NO *KRI* 51](#_Toc478256012)

[3.8.2.1. MODEL 51](#_Toc478256013)

[3.8.2.2. CONTROLLER 52](#_Toc478256014)

[3.8.2.3. VIEW 54](#_Toc478256015)

[3.8.3. RESUMO DA APLICAÇÃO DO *FRAMEWORK* 56](#_Toc478256016)

[4. DIFICULDADES ENCONTRADAS 56](#_Toc478256017)

[5. CONCLUSÕES 58](#_Toc478256018)

[6. REFERÊNCIAS BIBLIOGRÁFICAS 59](#_Toc478256019)

# LISTA DE FIGURAS

[Figura 1 – Modelo *MVC* e exemplo de requisição 15](#_Toc478256020)

[Figura 2 – Painel de controle do XAMPP 19](#_Toc478256021)

[Figura 3 – Tela principal do phpDesigner 20](#_Toc478256022)

[Figura 4 – Tela de um fluxograma na notação BPMN no SoftExpert 21](#_Toc478256023)

[Figura 5 – Interface do Navicat conectada em dois bancos diferentes 22](#_Toc478256024)

[Figura 6 - Estrutura de classes do *framework* 24](#_Toc478256025)

[Figura 7 - Arquivo index do diretório raiz 24](#_Toc478256026)

[Figura 8 - Diretório raiz 25](#_Toc478256027)

[Figura 9 - Arquivo de configuração 26](#_Toc478256028)

[Figura 10 - Arquivo de script principal 26](#_Toc478256029)

[Figura 11 – Arquivo *htaccess* utilizado para a reescrita da *URL* 28](#_Toc478256030)

[Figura 12 - Visualização do navegador ao exibir a estrutura da *URL* 29](#_Toc478256031)

[Figura 13 - Definindo o modulo padrão do sistema 29](#_Toc478256032)

[Figura 14 - Tratamento da *URL* 29](#_Toc478256033)

[Figura 15 – Sistema de roteamento 30](#_Toc478256034)

[Figura 16 – Sistema de roteamento 31](#_Toc478256035)

[Figura 17 – Inserção de logo em uma *view* 32](#_Toc478256036)

[Figura 18 – Visitando a imagem pelo *browser* descrita com a *URL* gerada 32](#_Toc478256037)

[Figura 19 – Logo armazenada na pasta *assets* 33](#_Toc478256038)

[Figura 20 – Declaração da constante global *ASSETS* 33](#_Toc478256039)

[Figura 21 – Utilizando a constante *ASSETS* em uma *view* 34](#_Toc478256040)

[Figura 22 – Resultado no *browser* do uso da constante 34](#_Toc478256041)

[Figura 23 – Acessando arquivos via *browser* 35](#_Toc478256042)

[Figura 24 – Erro gerado caso a constante do arquivo não tenha sido definida 35](#_Toc478256043)

[Figura 25 – Condicional a ser inserida em todos os arquivos do *framework* 36](#_Toc478256044)

[Figura 26 – Declaração da constante necessária para acesso ao sistema 36](#_Toc478256045)

[Figura 27 - Classe *Core* 37](#_Toc478256046)

[Figura 28 – Classe *Controller* 38](#_Toc478256047)

[Figura 29 - Tratamento dos atributos do construtor do *Controller* 38](#_Toc478256048)

[Figura 30 - Função que permite carregar uma *view* 39](#_Toc478256049)

[Figura 31 – Arquivo que permite carregar uma biblioteca para uso 41](#_Toc478256050)

[Figura 32 Arquivo função que carrega um *model* 42](#_Toc478256051)

[Figura 33 – Função que adapta a classe de conexão ao banco 43](#_Toc478256052)

[Figura 34 – Construção da classe *DataBase* 44](#_Toc478256053)

[Figura 35 – Classe que trata a conexão ao banco de dados 45](#_Toc478256054)

[Figura 36 – Classe SQLBuilder que permitirá criar consultas *SQL* 46](#_Toc478256055)

[Figura 37 – Criação de um consulta usando o sqlBuilder 46](#_Toc478256056)

[Figura 38 – Função que executa a consulta no banco de dados 47](#_Toc478256057)

[Figura 39 – Função que permite escrever uma consulta manualmente 47](#_Toc478256058)

[Figura 40 – Estrutura de diretórios do *framework* 48](#_Toc478256059)

[Figura 41 – Estrutura de um módulo 48](#_Toc478256060)

[Figura 42 – Pagina *web* do relatório de não conformidades 50](#_Toc478256061)

[Figura 43 – Consultas de coleta de dados para gerar o *KRI* 52](#_Toc478256062)

[Figura 44 – Chamada das funções do model retornando os dados do *KRI* 53](#_Toc478256063)

[Figura 45 – Classificação dos dados extraídos 53](#_Toc478256064)

[Figura 46 – Realização dos cálculos estatísticos 54](#_Toc478256065)

[Figura 47 – A apresentação em *HTML* dos valores coletados no Controller 55](#_Toc478256066)

[Figura 48 – Estrutura de repetição dentro da view 55](#_Toc478256067)

# LISTA DE SIGLAS E ABREVIATURAS

BPM – Business Process Management (Gestão de Processos de Negócio)

BPMS - Business Process Management Suite (Suite de Gestão de Processos de Negócio)

CRUD - Create, Read, Update, Delete (Criar, Ler, Atualizar e Deletar)

CSS - Cascading Style Sheets (Linguagem de Folhas de Estilo)

HTML - HyperText Markup Language (Linguagem de Marcação de Hipertexto)

IDE - Integrated Development Environment (Ambiente de Desenvolvimento Integrado)

PDO - PHP Data Objects (Objeto de Dados do PHP)

PHP - Hypertext Preprocessor (Preprocessador de Hipertexto)

URL - Uniform Resource Locator (Localizador Uniforme de Recursos)

SGBD - Data Base Management System (Sistema de Gerenciamento de Banco de Dados)

SO – Sistema Operacional

SQL – Structured Query Language (Linguagem de Consulta Estruturada)

UML - Unified Modeling Language (Linguagem de Modelagem Unificada)

# RESUMO

Este trabalho tem como objetivo apresentar as atividades do estágio supervisionado na Amaggi, no desenvolvimento de um *framework* *MVC* escrito na linguagem *PHP*. Este *framework* visa remover algumas atividades rotineiras do programador de sistemas *PHP*, agilizando assim a entrega das tarefas e priorizando a sua atenção com as regras de negócios. Assim como outros *frameworks* do mercado, também faremos a segregação de módulos, dessa forma, a manutenção dos sistemas criados sob o *framework* será de simples e de fácil migração. Será analisada aqui as rotinas fundamentais da estrutura *MVC* necessárias. Veremos a criação das classes, suas delimitações e os canais de comunicação entre elas para criar a ideologia do *MVC* no *framework*. Por fim analisaremos a migração e aplicação de uma página com *PHP* puro para a estrutura do *framework* criado fazendo uso das classes afim de atender toda a metodologia *MVC* e o uso de bibliotecas para acesso a banco de dados sem o contato direto a sintaxe de consultas *SQL*.

# INTRODUÇÃO

O grupo Amaggi atua no mercado do agronegócio produzindo, comprando, vendendo e exportando soja, milho e algodão. Contando com fabricas, escritórios, fazendas e armazéns em 5 países e matriz em Cuiabá. Com a crescente demanda de clientes, houve a necessidade de os acionistas garantir alguns certificados nacionais e internacionais, afim de garantir a procedência sustentável e a qualidade dos produtos da empresa. Sendo assim foram contratadas agencias certificadoras que possibilitaram a obtenção de algumas certificações nacionais e internacionais.

A agencia certificadora uma auditoria, sem qualquer ligação com a Amaggi, monitora o trabalho dos colaboradores do grupo, que é chamado de auditoria externa. Esta auditoria externa lista os pontos em que a empresa peca no cumprimento das diretrizes interna e aponta estes erros, que chamamos de “não conformidades”.

Estas não conformidades são inseridas em um sistema onde através de um processo modelos de processos conseguimos gerir e acompanhar a tratativa de cada não conformidade de forma singular, mas neste este sistema sua forma de acompanhar as tratativas de forma geral não é simples ou não é possível por limitações de seus idealizadores.

Tendo em vista estas limitações, ficou decidido que uma nova ferramenta para acompanhamento das não conformidades de forma ampla com geração de gráficos e relatórios deveria ser desenvolvida, baseando-se nos dados adquiridos pelo sistema citado, e refletindo nada mais que a realidade ali contida.

Assim, fez-se a necessidade de desenvolver um segundo sistema pequeno e de fácil manutenção que conecte com o ambiente do de tratativa das não conformidades através do banco de dados (somente leitura) que consiga gerar gráficos e relatórios gerencias. Mas a criação e manutenção de um sistema, mesmo que pequeno, demanda tempo e conhecimento do programador.

Como solução para este caso, iniciamos o desenvolvimento de um *framework* *web*, contendo bibliotecas, que possibilite aliviar a sobrecarga relacionada a atividades comuns do desenvolvedor, agilizando e focando o programador nas regras de negócios, otimizando e agilizando resultados.

# REVISÃO DE LITERATURA

## LINGUAGENS DE DESENVOLVIMENTO *WEB*

O processo de desenvolver páginas *web* é diferente do processo de desenvolver aplicações para ambientes *desktop*, em muitos aspectos. Uma aplicação *desktop* por exemplo, ao ser executada, permanece aberta na memória principal, e permite interagir com o usuário ou com o sistema de tempos em tempos mesmo de forma ociosa, o mesmo não acontece com uma aplicação *web*. Toda e qualquer interação feita pelo usuário à interface gráfica é realizada através de requisições. Toda requisição deve ser recebida, interpretada e um resultado deve ser gerado e entregue ou não ao usuário. Na prática, quando o usuário digita o endereço de um site no navegador, ele está fazendo uma requisição àquele site que interpreta a solicitação e gera uma saída que será renderizada no computador do usuário que solicitou.

Outro aspecto diferente e um dos mais problemáticos e que geram uma dificuldade no desenvolvimento é a não proteção contra o uso indevido de código, em outras palavras, dentro de um único arquivo, é possível colocar o código que define o funcionamento da página, e também a interface com o usuário. Em sistemas pequenos, isso pode parecer benéfico, uma vez que tudo se concentra em um arquivo, mas em sistemas grandes, isso pode ser bastante prejudicial.

Linguagens como *C#*[[1]](#footnote-1), por ter um ambiente de desenvolvimento integrado (IDE) própria, e uma estrutura muito bem definida, existe todo um aparato para proteção. Nas páginas *HTML* em *C#* existem apenas referências às variáveis geradas no corpo de código do sistema, e no código sistema não é possível exibir nada na interface com o usuário. Para um sistema grande isso gera uma estrutura organizada do código.

## O *FRAMEWORK*

Segundo Mattsson (2000, p. 07) um *framework* é um projeto reutilizável de todo ou parte de um sistema que é representado por um conjunto de classes e a maneira como suas instâncias interagem entre si.

Nosso objetivo então é focar na reutilização, reduzindo o retrabalho e maximizando resultados. Na orientação a objetos e o uso de bibliotecas vemos o início dessa filosofia de desenvolvimento.

## O MODELO *MVC*

Segundo Pitt (2012, p. 01),

“MVC (Model-View-Controller) é um padrão de projeto de software construído em torno da interligação de três tipos de componentes, em uma linguagem de programação como o PHP, muitas vezes com um forte foco em programação orientada a objetos.”

Trata-se de uma divisão bem definida de código e cada divisão da aplicação fica separado das outras que se comunicam por um canal bem definido. É possível definir 3 camadas que em teoria não devem interagir livremente no espaço uma das outras. Contudo devem existir uma via de comunicação para que o sistema funcione.

Assim podemos citar a página de código onde podemos definir o funcionamento do sistema em si; e a interface com o usuário, onde criamos uma interface amigável para uso do sistema, livre de código e da necessidade de conhecimento da linguagem usada para seu desenvolvimento.

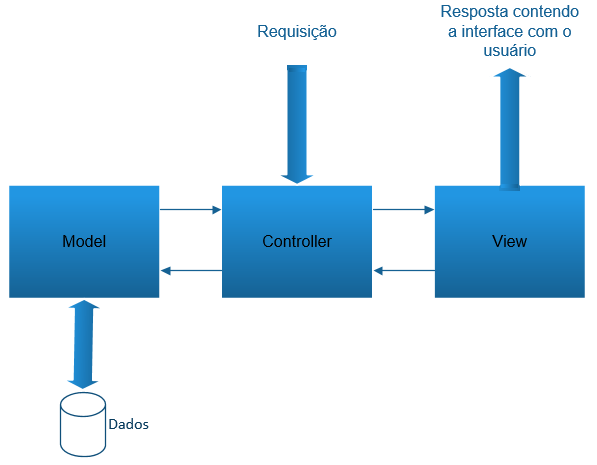
Essa é a divisão padrão de páginas *web*. O código que define o funcionamento do sistema não é visível ao usuário, e este vê apenas a interface gráfica. A página de código ainda pode ser separada e 2 camadas internas, uma camada de controle e uma camada de acesso a dados. Colocando todas em um único nível, ficando:

* Acesso a dados: dispõem bibliotecas e métodos para acessar dados independente da forma de armazenar (arquivo, sessão, banco de dados, integração com outros sistemas, etc.);
* Interface com o usuário: onde criamos uma interface amigável, para uso do sistema, livre de código e da necessidade de conhecimento da linguagem usada para seu desenvolvimento;
* Controle: ambiente onde os dados são selecionados, as decisões são tomadas e a saída é gerada.

No *PHP*, que será abordado aqui e utilizado no desenvolvimento do *framework*, esta separação não existe nativamente, o que pode tornar o código do sistema caótico e dificultando a leitura por outros programadores, manutenção e/ou até a documentação. Esta segregação não existe naturalmente no *PHP*, mas o modelo de padrão de projetos *MVC*.

O *MVC* define as três camadas citadas como *MODEL* (acesso aos dados), *VIEW* (interface) e *CONTROLLER* (controlador). O *model* e a *view* não conversam entre si sem passar pelo *controller*, como ilustra a Figura 1. Todas as requisições do usuário devem ser feitas ao *controller*, que deve processar as informações consultando ou não o *model*, e passar o resultado para a *view*. A *view* por sua vez deve montar o *HTML* e exibir ao usuário.

Figura 1 – Modelo *MVC* e exemplo de requisição



## *FRAMEWORKS* *MVC* ATUAIS

Existem uma variedade de *frameworks* *PHP* hoje no mercado, que podemos citar o *Zend* (<https://framework.zend.com/>), *Laravel* (<https://laravel.com/>), *CakePHP* (<https://cakephp.org/>), dentre outros. Para todos a essência do *MVC* é nativa. Suas diferenças se consolidam nos modos de uso, bibliotecas, classes nativas, padrão de reescrita de *URL*, etc., mas a principal de todas elas, e a que é crucial para a escolha de um *framework*, principalmente aos ingressantes no meio de desenvolvimento *web* é a curva de aprendizagem.

A curva de aprendizagem remete a ingressão em *frameworks* mais simples, enquanto os mais completos acabam ficando para os profissionais mais experientes ou jovens mais curiosos. Posteriormente os profissionais que optaram pelos mais simples acabam não se adaptando aos *frameworks* mais completos seja pela dificuldade de adaptação a nova estrutura ou porque acham que determinado *framework* é demasiadamente complicado.

Alguns *frameworks* trazem ferramentas que auxiliam na gerencia do processo de desenvolvimento como integração com sistemas de versionamento ou backup, através de acesso direto a comandos do sistema operacional (SO). Isso influencia diretamente na curva de aprendizagem, uma vez que o *framework* nos fornece uma gama de ferramentas uteis e outras necessárias (em certas ocasiões) para o funcionamento.

Todos os *frameworks* concorrentes do mercado possuem documentação, umas mais completas que as outras, e algumas poucas opções no português. Todas as documentações mostram como se usar e aplicar determinado *framework* na criação de sistemas, mas nenhuma mostra o funcionamento interno dele.

A título de exemplo citamos o Laravel que é de código aberto, e tem seu código fonte disponível para download. A documentação não trata do explicitamente do funcionamento do núcleo do *framework*, mas apenas como aplica-lo no desenvolvimento de aplicações. Caso ele não atenda a uma determinada regra de negócio, nada pode ser feito.

Diante deste impasse, ficou resolvido a criação um *framework* personalizado, totalmente novo, e que se conhecendo toda a estrutura desde a chamada do arquivo principal, todo o funcionamento interno, até a geração de consultas *SQL*, o *framework* pode ser modificado e evoluir a fim de atender a todas as especificações do negócio.

Nosso *framework* *MVC* será baseado sob os conceitos da orientação a objetos, então toda a sua abstração é imprescindível para o desenvolvimento e conhecimento do seu funcionamento.

# MATERIAS, TÉCNICAS E MÉTODOS

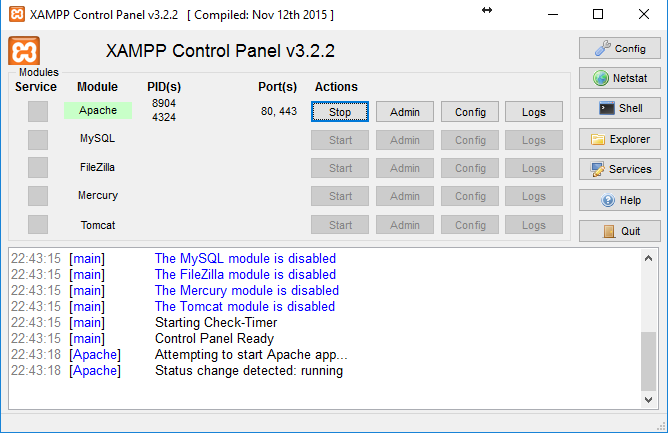
A proposta do projeto é criar um *framework* que facilite e agilize o processo de desenvolvimento de sistemas baseado em *PHP*. Para auxilio nesta etapa, são necessárias algumas ferramentas que forneça o mínimo de suporte. Todas as funções das aplicações desta sessão foram necessárias para o sucesso do projeto, mas ainda assim, todas elas podem ser substituídas por aplicações concorrentes.

## WAMPP

O WAMPP (<https://www.apachefriends.org/>) é um pacote contendo aplicações já integradas, suficiente para implementar um servidor *web* funcional, sem a necessidade de configurar vários arquivos separadamente. Basicamente ele é composto pelo Apache que segundo Kabir (2002, p. 03, 04) é um servidor de páginas *web* de código aberto, multiplataforma e o mais usado no mundo, e o *PHP* que segundo Doyle (2000, p. 03) é uma linguagem de programação para criar páginas *web* de forma dinâmica mas que precisa de servidor *web* para pleno funcionamento. Além destes dois principais *softwares*, o WAMPP possui o MySQL (<https://www.mysql.com>), phpMyAdmin (<https://www.phpmyadmin.net/>), OpenSSL (<https://www.openssl.org/>), FileZilla FTP Server(<https://filezilla-project.org/>), entre outras ferramentas complementares. Na Figura 2 vemos a tela do painel de controle do XAMPP. Nela podemos instalar as aplicações ocmo serviço, ou abrí-las como um programa desktop comum.

No painel de controle do XAMPP é possível fazer as operações básica dos processos, abrir os logs, arquivos de configuração, abrir as telas de administração e instalar os serviços de cada aplicação nele contida. Nenhum processo é iniciado com o sistema, a não ser que os serviços sejam instalados, dessa forma ele é um conjunto de aplicações pronta para atender ambientes de desenvolvimento, teste e produção.

Figura 2 – Painel de controle do XAMPP

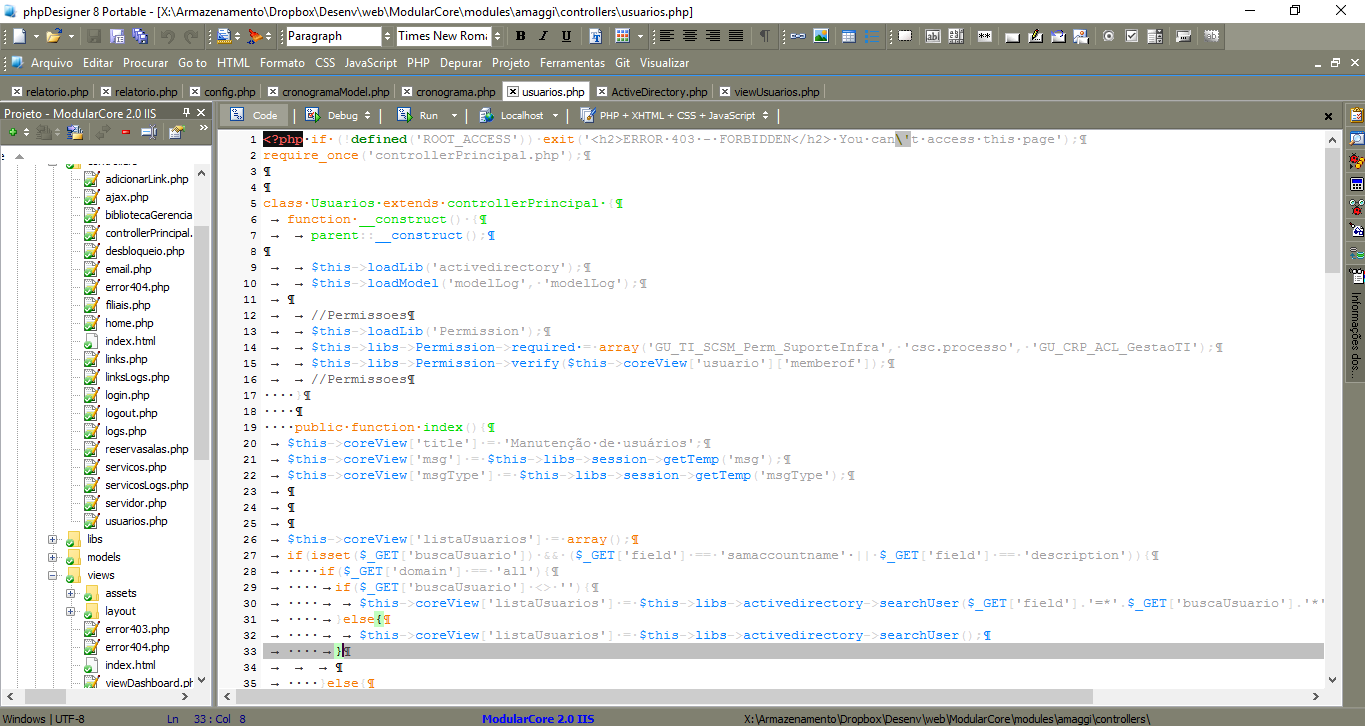


## PHPDESIGNER

O phpDesigner é uma IDE de desenvolvimento *web* voltado diretamente para as linguagens *PHP*, *HTML*, *javascript*, *CSS* e scripts *SQL*. Ele facilita o desenvolvimento de sistemas *web* oferecendo dicas (*balões*) de métodos e atributos da linguagem e orientação a objetos. Fornece também uma coloração que favorece a compreensão dos elementos do código, como funções, variáveis, *string*, comentários, etc. E altera a coloração de setores de linguagens diferentes no mesmo arquivo, ajudando na identificação de blocos de códigos *javascript* dentro de um *HTML* por exemplo e também colorindo as palavras chaves facilitando a visualização do arquivo como na Figura 3.

Além de ter um gerenciador de arquivos ele ainda conta com um interpretador *PHP* interno que facilita o processo de *debug* [[2]](#footnote-2)e integração ao *Git*[[3]](#footnote-3). Possui também um buscador poderoso de texto em arquivos e com ele é possível buscar um determinado termo dentro de todos os arquivos do projeto. E por fim tem a função chamada ”*PHP* *Code* *Beautifier*” (embelezador de código *PHP*) que faz indentação automática e organiza estruturas de código como condicionais e repetição.

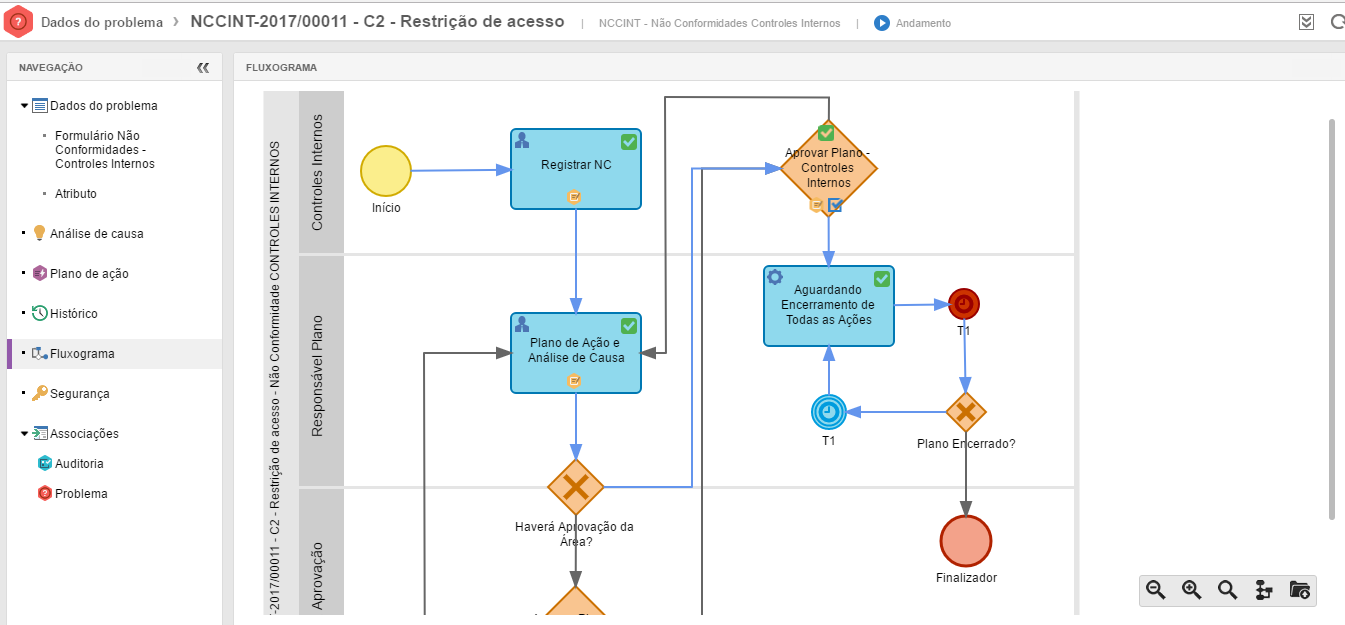
Figura 3 – Tela principal do phpDesigner



## SOFTEXPERT

O SoftExpert *Excellence* *Suite* (<https://www.softexpert.com.br/>) é uma suíte de soluções para automação de processos BPM da empresa SoftExpert que é um sistema BPMS (Business Process Management Suites) fundada em 1995. Possui módulos de documentos, desempenho, *Business* *Intelligence*, problemas, incidente, risco dentro vários outros. Nele conseguimos gerenciar o tratamento de não conformidades apontadas pela auditoria, acompanhando, delegando atividades, criando e executando planos de ações corretivas e preventivas, e aplicação de metodologias como “5 porquês” e “diagrama de causa e efeito”. Na Figura 4 vemos a notação BMP e o fluxograma do processo de tratamento de não conformidades.

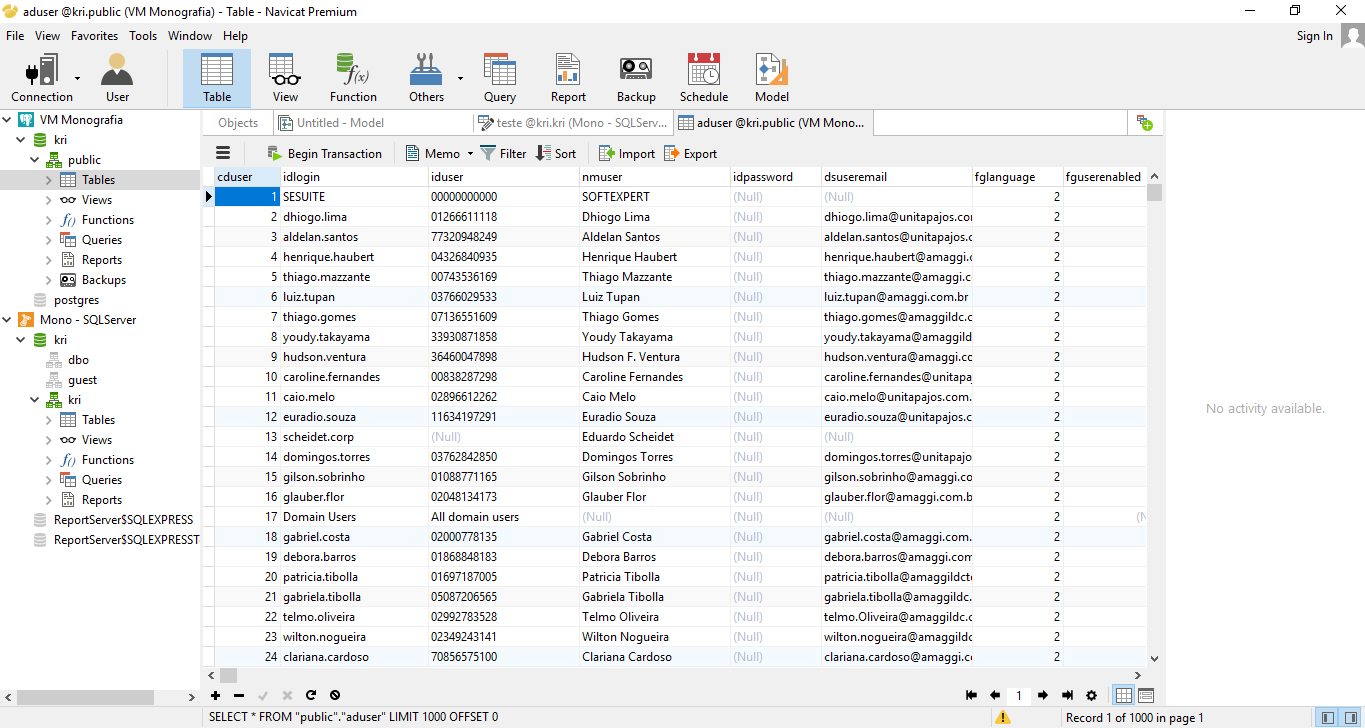
Figura 4 – Tela de um fluxograma na notação BPMN no SoftExpert



## NAVICAT

O Navicat (<https://www.navicat.com/>) é um multicliente de banco de dados relacional (SGBD) criado pela empresa PremiumSoft (<https://www.navicat.com/company/aboutus>) fundada em 1999. Ele possui a capacidade de gerenciar bancos Oracle, Postgres, SQL Server, MySQL, SQLite e MariaDB. Além de conseguir conectar com mais de um banco diferente ao mesmo tempo ele tem um sincronizador de estrutura entre bancos de dados. Possui um modelador de diagramas UML, além de ter um gerenciador de *views*, *function* e *triggers*, backup e um migrador entre bandos de dados e fabricantes diferentes. Possui versões *Windows*, *Linux* e *Mac*. *Software* pago e para este trabalho foi utilizada sua versão para testes. Na Figura 5 vemos a tela do programa conectada a dois bancos de dados de diferentes desenvolvedoras ao mesmo tempo, onde poderíamos transferir dados de um banco para outro.

Figura 5 – Interface do Navicat conectada em dois bancos diferentes



## POSTGRESQL

O PostgreSQL ou comumente chamado de Postgres que segundo POSTGRESQL (2017, p. xvii) é um sistema gerenciador de banco de dados objeto-relacional. É um SGDB de código aberto criado pela PostgreSQL Global Development Group. Disponível para os sistemas Windows, Linux e Unix, e com mais de 15 anos de desenvolvimento ele é bastante estável, permite criar bancos de dados com terabytes de informações, de download livre e totalmente sem custo. Foi escolhido para este trabalho por ser o mais robusto dos SGBD de uso gratuito. Ele não possui interface gráfica, assim como outros SGBD, e originalmente é controlado por linha de comando. Para facilitar a administração dele é necessário um aplicativo gerenciador como o Navicat descrito na sessão 2.1.4.

## ORACLE

O Oracle Database (<https://www.oracle.com>) é o SGBD da Oracle Corporation. Lançado na década de 70, hoje é líder de mercado e considerado por muitos o SGBD mais robusto e estável. Possui 3 versões para uso corporativo e uma versão gratuita que limita uso de dados e quantidade de usuários conectado para testes de desenvolvimento. É o SGBD onde o SoftExpert guarda os dados.

# RESULTADOS

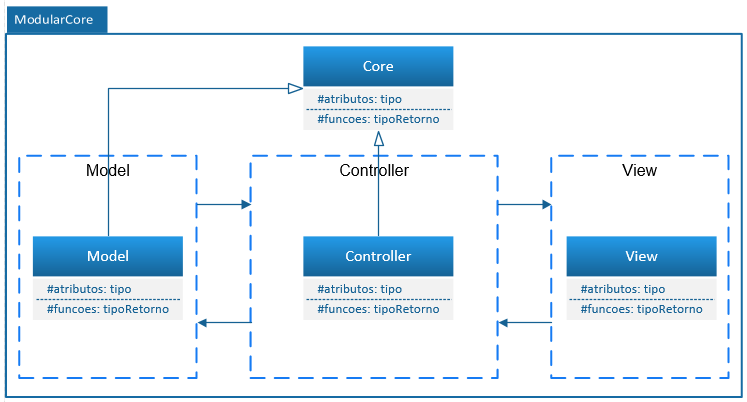
## INTRODUÇÃO AO FUNCIONAMENTO DO *FRAMEWORK*

Para entendermos o funcionamento básico do *framework*, precisamos entender primeiramente como as camadas trabalham. Elas precisam executar as funções para qual foram criadas, mas não podem permitir que o desenvolvedor crie ou use ferramentas dentro de camadas que não foram desenvolvidas para tal. Não é interessante que o programador faça acesso ao banco diretamente do *controller* ou da *view* por exemplo.

Para conseguirmos estabelecer a facilidade de uso das ferramentas e bibliotecas do *framework* e os limites de funcionamento de cada parte dele, precisamos criar uma estrutura de classes que permita isso. Como vermos na Figura 6, essa estrutura simples permite que as camadas *model*, *view* e *controller* contenham os atributos e funções que permitirão seu funcionamento completo e facilidades sem interferir no funcionamento do outro. As classes do desenvolvedor estenderão essas classes principais afim de herdar as características e limitações de suas superclasses. Assim, mesmo que o usuário queira executar funções de *controller* dentro do *model*, ou vice-versa, será impedido de utilizar as funções do outro e estará limitado a sua respectiva estrutura e funcionamento. Como regra na orientação a objetos as classes podem herdar as caracterizas de apenas uma classe isso consolida as afirmações de características únicas de cada camada, e por esse motivo não utilizamos interface, pois uma classe pode implementar as caracterizas de uma ou mais interfaces.

Quando iniciarmos o desenvolvimento do *framework*, não iremos instanciar nenhuma classe. Todas as classes descritas aqui devem ser classes abstratas. Ocorrerá que o desenvolvedor criará um *controller* ou um *model* que estenderá a classe *Controller* ou *Model* respectivamente. A classe *controller* criada poderá chamar funções de classes *model* (assim como na definição de *MVC*), e carregar arquivos da *view*. Na sessão 3.2 vamos iniciar a execução dos scripts escolhendo a classe correta a ser instanciada e a criação dela na prática.

Figura 6 - Estrutura de classes do *framework*



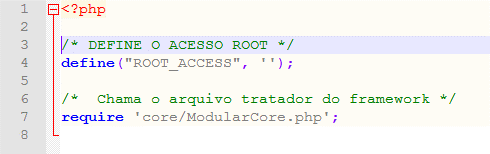
## INDEX.PHP E O INICIO DO PROJETO

O *index.php* é o arquivo padrão inicial de um sistema projetado em *PHP*. Por padrão o Apache busca o arquivo *index.html*, caso não obtenha sucesso, ele busca o arquivo *index.php* junto com o interpretador do *PHP* (caso o *PHP* esteja integrado e devidamente configurado). Em outras palavras o *index.php* é a porta de entrada para um sistema *web* baseado em *PHP*. Isso não será diferente do *framework* desenvolvido aqui.

No diretório raiz do projeto será criado um arquivo *index.php*, e todas as requisições feitas em nosso sistema passará primeiramente por este arquivo, que carregará um script principal que se encarregará de tratar a solicitação e tomar o caminho correto dentro do sistema.

O usuário não verá que o sistema passa primeiramente por este arquivo, mas ele verá *URL* estilo diretório. Dessa forma conseguiremos proteger os arquivos do nosso sistema de olhos curiosos. Na Figura 7 vemos a construção de arquivo.

Figura 7 - Arquivo index do diretório raiz



No arquivo *core/ModularCore.php* vamos iniciar o tratamento das requisições. Iniciamos com a definição de um *namespace* do projeto afim de interferir o mínimo possível nos nomes das classes, funções e constantes do projeto do desenvolvedor. Tendo em vista que o projeto se encontra em uma estrutura de diretórios complexa, extensa e que não segue um padrão, iremos utilizar algumas constantes para facilitar a localização dos arquivos do projeto dentro do SO, caso contrário, precisaríamos usar funções nativas do *PHP* e algumas rotinas a cada vez que precisarmos apontar um arquivo no disco.

## MODULARIZAÇÃO

Os *frameworks* atuais do mercado trabalham com a ideia de módulos. Estes módulos funcionam como partes de um sistema complexo (como modulo de vendas, almoxarifado, RH, etc.) ou até pequenos sistemas distintos. Como é uma premissa facilitar a vida do desenvolvedor, fazemos a varredura de módulos disponíveis no projeto. O desenvolvedor irá criar seus projetos no diretório de módulos, tendo a ideia de um sistema modular. Assim se um determinado modulo estiver presente no diretório do projeto, ele estará disponível para uso, sem a necessidade de declarar ele em um arquivo de configuração. Removendo este modulo do diretório, o modulo simplesmente deixa de existir no sistema e passará a retornar uma página de erro caso o usuário tente o acesso. Para isso vamos criar um diretório chamado *MODULES* na raiz do projeto, e outro chamado RELATORIO dentro dele.

Figura - Diretório raiz

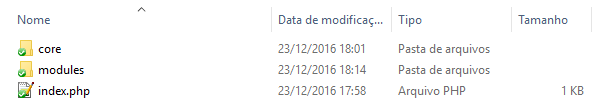


Figura 9 - Arquivo de configuração

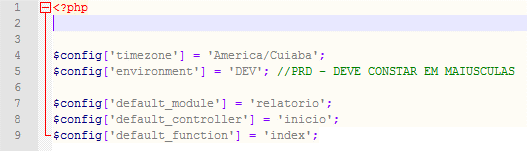
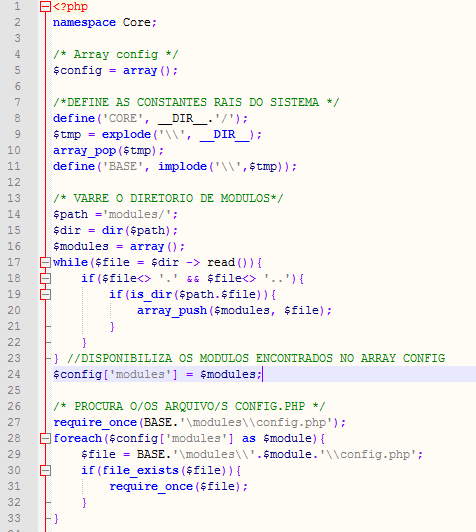


Figura 10 - Arquivo de script principal



As configurações relativas a banco de dados, uso de bibliotecas, classes, entre outras podem variar muito de módulo para módulo, como o desenvolvedor preferir, mas ainda há a possibilidade de um sistema grande que todos os módulos compartilhem das mesmas configurações. Sendo assim, o sistema pode ter configurações compartilhadas e configurações particular para cada módulo. Caso a configuração particular do módulo seja encontrada, o *framework* fará uso dela. Caso o arquivo de configuração particular não seja encontrada, ele fará o uso das configurações comuns a todos os módulos. Então precisamos inicialmente criar um arquivo de configuração (por conversão sendo *config.php*) no diretório raiz dos módulos (*core/modules/config.php*).

No arquivo *config.php* vamos definir uma variável do tipo *array* e algumas posições que faremos uso posteriormente quando chegarmos na estrutura das classes do *framework*.

## ENDEREÇOS AMIGÁVEIS E A REESCRITA DE *URL*

Para nosso projeto iremos acessar um determinado modulo que escolhermos, seja o modulo padrão do projeto, ou um modulo especifico. Para isso ao digitar [*http://dominio/*](http://dominio/)queremos acessar o modulo padrão, e caso haja algo após, descreve o modulo especifico, chamando assim um script especifico do nosso módulo. Para isso utilizaremos a reescrita de *URL*.

A reescrita de *URL* permite utilizarmos endereços tais como [*https://dominio/noticias/empresas/google-pixel/novo-lancamento-de-celular*](https://dominio/noticias/empresas/google-pixel/novo-lancamento-de-celular)*.* Este tipo de *URL* apresenta o caminho do site ou sistema de uma forma elegantes e permite um fácil entendimento pelo usuário. Ele não acessa a estrutura de diretórios diretamente, mas sim uma estrutura de módulo/*controller*. Para isso precisamos configurar um arquivo chamado *.htaccess[[4]](#footnote-4),* que é um arquivo que será interpretado pelo Apache, e fornecerá uma *URL* escrita de forma diferente ao *PHP* da que é apresenta no *browser* para o usuário.

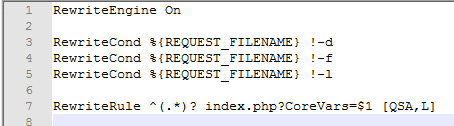
* O usuário verá: [*https://dominio/noticias/empresas/google-pixel/novo-lancamento-de-celular*](https://dominio/noticias/empresas/google-pixel/novo-lancamento-de-celular)
* O *PHP* entenderá: [*https://dominio?variavel=noticias/empresas/google-pixel/novo-lancamento-de-celular*](https://dominio?variavel=noticias/empresas/google-pixel/novo-lancamento-de-celular)

Dentro do *PHP* essa *URL* chegará por uma variável global chamada *$\_GET*[[5]](#footnote-5), e podemos trata-la. Vamos quebrar essa variável pelas barras e assim podemos ter um *array* com várias posições indicando cada trecho entre as barras.

## O FUNCIONAMENTO DA REESCRITA DE *URL*

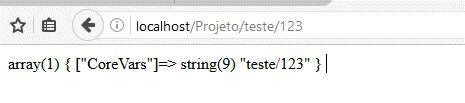
A reescrita de *URL* irá traduzi-la real em uma URL com um ou N parâmetros via requisição *GET*. A *URL* para o usuário é escrita de uma determinada forma, mas para o sistema de forma global ela é escrita de outra, assim como descrito na sessão 3.4. Sendo assim, podemos definimos o arquivo *.htaccess*.

Figura 11 – Arquivo *htaccess* utilizado para a reescrita da *URL*



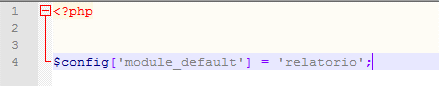
Basicamente neste arquivo de configuração do Apache definimos que tudo que estiver após o endereço do domínio, chegará ao *PHP* via *GET* como uma variável global denominada *“$\_GET[‘CoreVars’]”* (o *PHP* interpreta os parâmetros via *GET* e *POST* sendo como um *array*). Assim assumimos um *array* chamado *$CoreVars* com N posições. Nesse *array* vamos atribuir todos os dados da variável global *“$\_GET[‘CoreVars’]”*. Será necessário para trabalharmos com o sistema de roteamento posteriormente.

Figura 12 - Visualização do navegador ao exibir a estrutura da *URL*



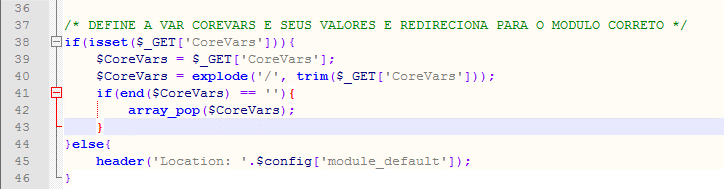
Caso seja digitado apenas o *http://dominio/*, precisamos escolher um modulo padrão. Então vamos utilizar a posição nomeada de *‘module\_default’* do *array* de configuração*.* Precisamos dessa configuração para informar ao sistema que quando o usuário não solicitar o acesso a um determinado modulo, ele deverá seguir por um caminho padrão.

Figura 13 - Definindo o modulo padrão do sistema



Assim, sempre que tentarmos acessar a raiz do domínio, seremos redirecionados automaticamente para [*http://dominio/relatorio*](http://dominio/relatorio), mas ao digitarmos algo depois do domínio, conseguiremos acessar o modulo escolhido.

Figura 14 - Tratamento da *URL*

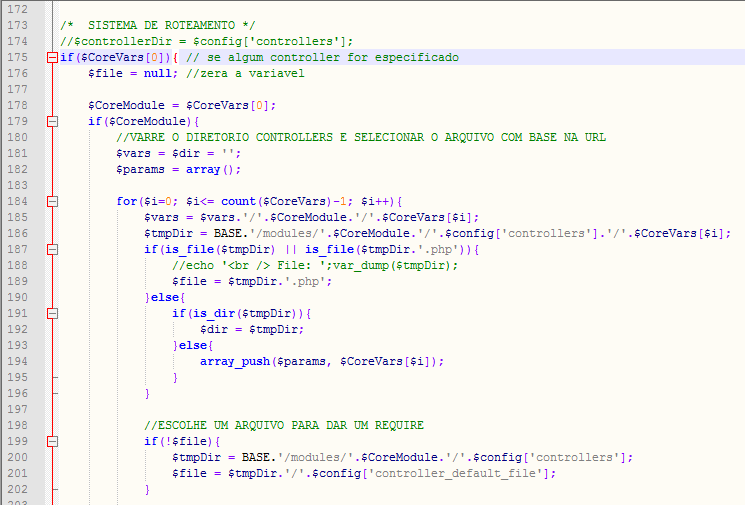


Quando este acesso acontece, não estamos acessando o diretório no servidor, mas sim acessando o *index.php* passando algo como parâmetro. Isso irá permitir que o usuário visitante não consiga navegar na estrutura de diretórios do servidor, mas sim por uma estrutura virtual criada pelo desenvolvedor, característica que reflete parte da segurança do *framework* abordado na sessão 3.5.

## SISTEMA DE ROTEAMENTO

De forma simples, o sistema de roteamento interpretará a *URL* e irá direcionar a requisição para o *controller* correto. Já dentro do *controller*, o script iniciará uma determinada função ou caso não tenha sido informada, será chamada à função padrão descrita no arquivo de configuração, trabalhando em conjunto com a reescrita de *URL*. Ainda não definimos um *controller*, mas basta nosso script chamar o arquivo desejado mesmo que seja apresentado um erro, pois sabemos que este será executado assim que existir.

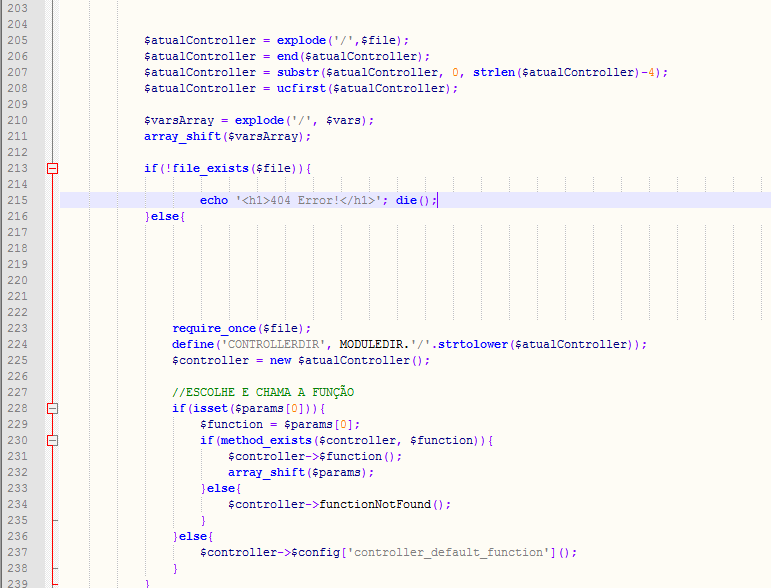
Figura 15 – Sistema de roteamento



Por convenção iremos entender como módulo principal a posição *‘default\_module'* do arquivo de configuração escolhido e *controller* principal aposição *‘default\_controller'*. Dentro do diretório *modules/relatorio/controllers controllers* podemos ter tantos *controllers* quanto necessários para suprir as necessidades do sistema.

Na Figura 15 o sistema de roteamento irá avaliar o *array* *$CoreVars* gerado anteriormente pelos parâmetros passados via *URL* e entender que a primeira posição será o modulo a ser acessado, o segundo parâmetro será o *controller* do modulo, e o terceiro parâmetro a função do controller. Os demais parâmetros poderão ser definidos pelo desenvolvedor.

Figura 16 – Sistema de roteamento



Na linha 201 do arquivo *ModularCore.php* é definido um arquivo para ser agregado ao script e na linha 223 o arquivo escolhido é agregado (pela função *require\_once()*[[6]](#footnote-6)), a classe do *controller* escolhido é instanciada na linha 225 e finalmente na linha 237 a função é chamada.

Caso o usuário tenha definido uma *URL* que não corresponda a um controller, a exceção é tratada na linha 213. Será apresentado um erro 404 e a função global *die()* [[7]](#footnote-7)é chamada. O tratamento da exceção da chamada da função do *controller* fica na linha 230, e caso a função não exista será chamada a função *functionNotFound()* que simplesmente chamará uma mensagem de erro através da implementação do *controller*.

## DEPENDENCIAS *HTML*

Ao criar o sistema de roteamento onde o que digitamos na *URL* significa algo diferente para o *framework*, criamos também um problema com o referenciamento de dependências do *HTML* (como *CSS*, jasvascript) e imagens. Ao tentar referenciar algum objeto dentro da estrutura de diretórios em que o *framework* se encontra assim como ilustra a Figura 17, os primeiros diretórios irão cair na regra do roteamento, onde o primeiro, o *framework* interpretará como sendo o modulo a ser acessado, o primeiro subdiretório sendo o *controller*, e o segundo sendo a função do *controller*.

Figura 17 – Inserção de logo em uma *view*

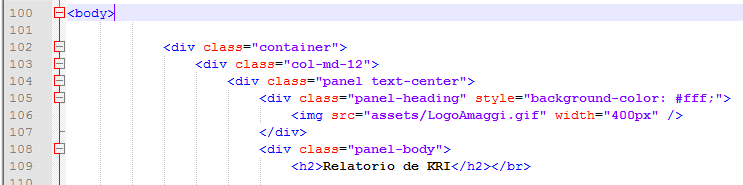
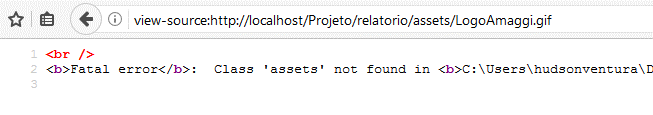


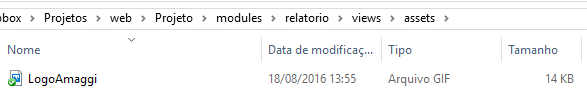
Figura 18 – Visitando a imagem pelo *browser* descrita com a *URL* gerada



Sendo assim estaremos gerando um link de forma errada e como resultado o *framework* interpreta nossa referência como vemos na Figura 18. O *framework* acessou o modulo relatório e tentou instanciar a classe *assets* como sendo um *controller*. Totalmente coerente para o funcionamento do roteamento.

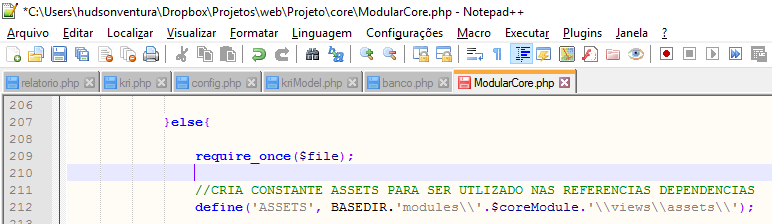
Para resolver tal características, teremos que criar uma exceção para este tipo de referência. Primeiramente precisamos definir um diretório dentro do modulo para armazenarmos estes arquivo, e optamos pelo diretório de *views* (*modules/moduloescolhido/views/assets*), justamente por se tratar de arquivos referentes a visualização da interface gráfica. Criamos então o diretório especificado e guardamos a logo da empresa.

Figura 19 – Logo armazenada na pasta *assets*



Precisamos então criar um link de referência global para ser utilizado nas *views*. Este link irá trazer o caminho padrão do diretório *assets*. Na Figura 20 linha 212, fazemos a criação da referência. Assim, toda vez que a constante global *ASSETS* for utilizada, ela estará apontando para o diretório *modules/moduloescolhido/views/assets*, e ignorando o funcionamento padrão do roteamento.

Figura 20 – Declaração da constante global *ASSETS*



Dessa forma teremos que utilizar a constante em todas as nossas referencias como na linha 106 da Figura 21, obtendo o resultado da Figura 22. O *browser* conseguiu interpretar o caminho correto da imagem.

Figura 21 – Utilizando a constante *ASSETS* em uma *view*

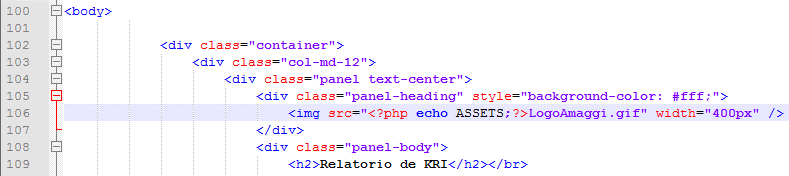
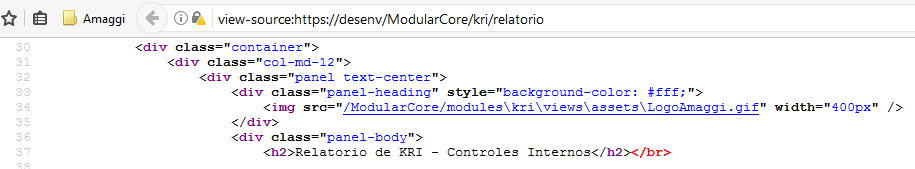


Figura 22 – Resultado no *browser* do uso da constante



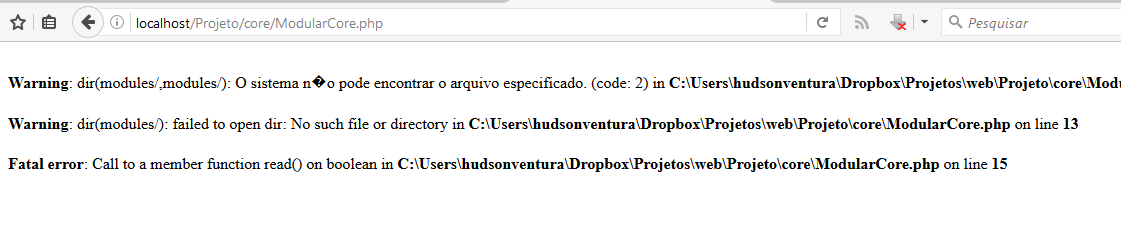
## INICIANDO A SEGURANÇA DO PROJETO

A segurança em sistemas computacionais é uma sessão bastante delicado. Muitas formas de ataques a sistemas são possíveis, e um sistema *web* também pode sofrer vários tipos de ataques. Este assunto poderia render um trabalho bastante extenso. Mas existe um detalhe que podemos resolver de maneira bastante simples e já incorpora um pequeno nível de segurança.

Em nosso projeto, é importante não permitirmos que o usuário acesse arquivos de configuração do sistema, e arquivos de script. Caso estes arquivos sejam acessados por pessoas mal-intencionadas, elas podem obter as credenciais do banco de dados, e roubar dados dos usuários. Precisamos nos preocupar com os vários tipos de ataques desde o início do projeto, e devem ser constantemente revisados afim de minimizar a possibilidade de sofrer um ataque. Não vamos nos preocupar em proteger o banco de dados, pois esse não é nosso objetivo no momento. Vamos nos focar em proteger a aplicação.

Já iniciamos a segurança da estrutura de diretórios através da definição da reescrita de *URL*, abordado na sessão 3.4, que dificulta que o usuário final liste os arquivos de um diretório, mas caso ele digite o endereço completo de um arquivo ele ainda irá conseguir acessá-lo. Isso não pode acontecer em hipótese alguma com os arquivos de configurações do sistema. Nele são guardadas as configurações de acesso ao banco de dados e um acesso a este arquivo pode gerar um transtorno muito grande para o administrador do sistema. E é este ponto que trataremos neste momento.

Figura 23 – Acessando arquivos via *browser*



Para resolver podemos requerer que o usuário tenha passado pelo script principal e que nele tenha definido uma constante ao acessar um arquivo qualquer, assim garantiremos que ele passará pelo script principal antes de acessar outros arquivos e impossibilitaremos que ele acesse um arquivo diretamente. Nesta etapa do projeto, ao acessar o arquivo *ModularCore.php*, podemos ver o resultado na Figura 23. Neste caso os erros do *PHP* são apresentados para o usuário pois não tem os dados essenciais para iniciar o script e os erros ainda não foram tratados.

Figura 24 – Erro gerado caso a constante do arquivo não tenha sido definida



Para evitar tal acesso, vamos definir a constante mencionado anteriormente no arquivo *index.php*, e uma condicionalem todos os outros arquivos do *framework*. Com isso, teremos o resultado da Figura 24. Por fim vamos adicionar o trecho de código da Figura 26 no arquivo *index.php* raizpara que ele possa acessar os arquivos projetos normalmente.

Figura 25 – Condicional a ser inserida em todos os arquivos do *framework*



Figura 26 – Declaração da constante necessária para acesso ao sistema



## ESTRUTURA DE CLASSES DO *FRAMEWORK*

Até o momento trabalhamos na execução do script inicial do *framework*, que escolhe o modulo e o *controller* que o usuário solicitou e instancia a classe *controller* estendida pela classe do desenvolvedor. Assim como descrito na Figura 6 nossa entrada para o sistema será pelo *controller*, e assim podemos acessar o *model* para acessar e tratar os dados e a *view* para retornar as informações para o usuário.

## A CAMADA CONTROLLER

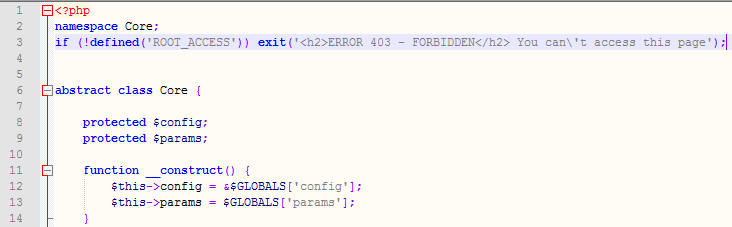
O *CONTROLLER* é a classe que receberá todas as requisições feitas pelo usuário e fará o processamento dessa requisição ao qual foi programado e gerará um retorno através de uma *view*. Em poucas palavras ele é a entrada de qualquer ação dentro do sistema. Nada será feito no sistema senão pelo *controller*.

Nossa camada de *controller* será uma classe abstrata que permitirá ser estendida pelas classes filhas do desenvolvedor, herdando suas características e limitações. Somente as classes reais poderão ser instanciadas e atender as requisições do usuário, de forma a atender a premissa de ser um *controller*. O *controller* não poderá executar funções de *model* ou *view*.

Vamos criar os arquivos *core/Core.php,* e nele declaramos a classe abstrata *Core* que vamos incluir todos os atributos e funções comuns a camada *controller* e *module.* Não incluiremos nesta fase a classe view pois ela tem recursos totalmente distintos das outras classes e trataremos posteriormente.

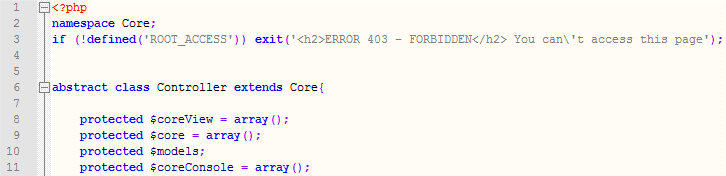
Os atributos mencionados na Figura 27 bem como a função *\_\_construct()* [[8]](#footnote-8)irá trazer as variáveis *$coreConfig* e *$params* contidas no script principal para a camada de classes do *PHP*. Isso auxiliará o desenvolvedor no momento de escrita dos seus *controllers* e *models*.

Figura 27 - Classe *Core*



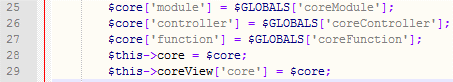
Criamos também o arquivo *core/Controller.php* e nele declaramos a classe abstrata *Controller* que estende a classe *Core.* Também criaremos alguns atributos para auxiliar o processo de desenvolvimento.

Figura 28 – Classe *Controller*



O atributo *$core* definimos um *array* com os dados básicos do módulo, *controller* e *function* do controller acessados no momento. Isso permitirá ao desenvolvedor tratar os acessos de uma forma global sem que ele tenha que declarar o real módulo que o usuário está acessando no momento. O atributo *$coreView* definimos um *array* que estará disponível na *view*.

Figura 29 - Tratamento dos atributos do construtor do *Controller*

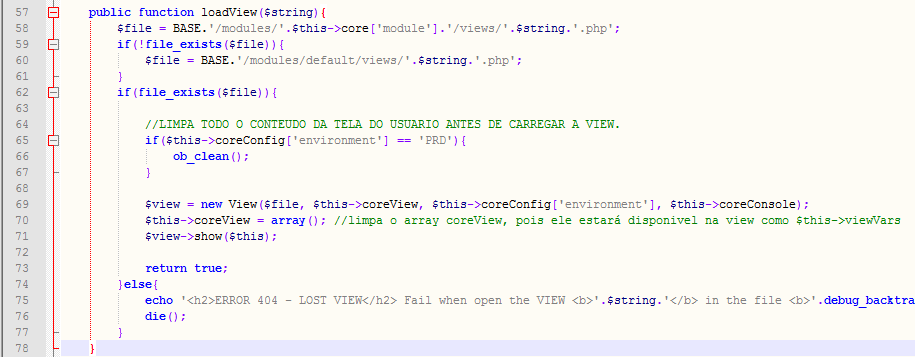


## A CAMADA VIEW

A camada view será responsável por exibir ao usuário o conteúdo processado pelo *controller* e pelo *model*. Criamos a classe *View* no arquivo *core/view.php* assim como as demais camadas. A premissa da *view* é não podemos permitir que o desenvolvedor acesse as funções e bibliotecas do *controller* e do *model*, mas também não podemos deixar que quaisquer funções de exibição de dados na tela sejam acessadas por eles (como usar a função *echo()*[[9]](#footnote-9) no *controller*). Para isso bastaria bloquear que a função *echo()* não seja disponível no *controller* e *model*, contudo a linguagem não nos permite isso.

O *PHP* trabalhar com o armazenamento dos dados que serão exibidos ao usuário em cache no servidor, e por padrão, só depois que toda a página é gerada o cache é descarregado para a tela do usuário. O *PHP* permite que descartaremos todos os dados da cache gerados até determinado momento. Então descartaremos essa *cache* antes que a *view* seja carregada. Assim estaremos forçando o programador a escrever os dados na tela ao usuário obrigatoriamente na *view*, uma vez que tudo antes do carregamento da *view* será descartado. Para tal utilizaremos a função *ob\_clean()*[[10]](#footnote-10)*.* Este será o invólucro de uso da camada. Mas isso vai contra a premissa de facilitar o processo de desenvolvimento, e para debugar o código será necessário utilizar funções como *var\_dump()* [[11]](#footnote-11)por exemplo. Utilizando a função *ob\_clean()* vamos acabar impedindo que o programador utilize o *var\_dump()* ou *XDebug*[[12]](#footnote-12). Então vamos precisar criar uma exceção que veremos no final desta sessão.

Figura 30 - Função que permite carregar uma *view*



A classe *View* será tratada em duas etapas: construção e exibição. Na construção da classe vamos receber o arquivo que deve ser incluído, as variáveis a serem exibidas na tela para o usuário, o tipo de ambiente e dados do console, todos passados pelo *controller* no construtor da classe *View*. Este será o canal de comunicação *controller->view*.

Na linha 69 passamos todos os dados necessários para que a *view* funcione corretamente. Caso o arquivo *$file* não exista, a classe *Controller* faz a tratativa e apresenta um erro 404 e aciona a função *die()* nativa do *PHP*. Junto do arquivo é passando o atributo (*array*) *$coreView* com todas as variáveis atribuídas no *controller* (uma para cada posição do *array*), que estarão disponíveis na camada *view* como variáveis, e por fim o atributo *$coreConsole*, que vamos aplicar posteriormente.

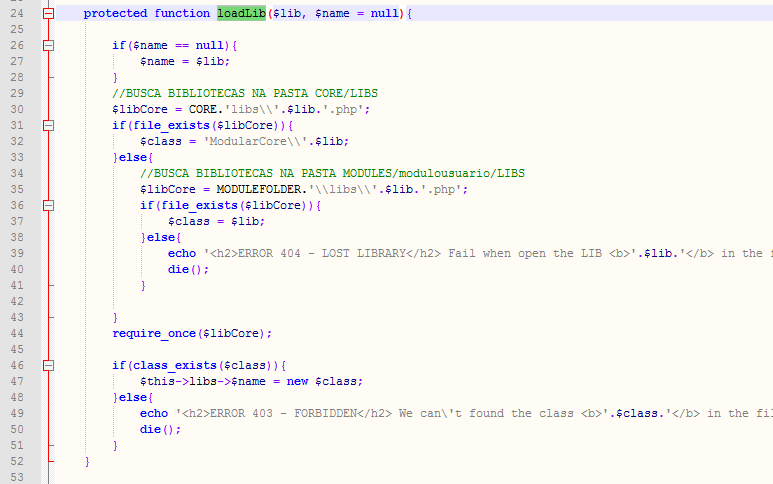
Na linha 65, tratamos a exceção do tipo de ambiente. Caso o ambiente seja de produção, vamos apagar tudo que foi gerado nas camadas anteriores a *view*. Caso o ambiente não seja o de produção, podemos deixar tudo o que foi gerado para ser apresentado na tela. Isso facilitará o processo de *debug*, mas também permitirá que o ambiente produtivo e os olhos dos usuários fiquem livre de códigos. Para isso basta definir o valor da posição ‘*environment'* do *array* de configuraçãodentro do arquivo de configuração, como definido na sessão 3.3.

## CARREGAMENTO DE BIBLIOTECAS

As bibliotecas nos fornecerão ferramentas para abstrair o uso de terminadas ferramentas como veremos na sessão 3.6.5. As bibliotecas estarão disponíveis para o programador utilizá-las dentro das camadas *controller* e *model*. Para ambos é necessário que sejam previamente criadas, ou utilizar uma biblioteca que já esteja presente no *framework*. Aqui vamos criar uma biblioteca para manipular o banco de dados da camada *model*.

Na classe *Core* criamos a função *loadLib()* que tratará de varrer as bibliotecas contidas em *core/libs* e *modules/relatório/libs* quando solicitadas. Na Figura 31 vemos o a função no arquivo *core/Core.php.* Devemos passar qual biblioteca deverá ser carregada e um nome. Nas linhas de 30 a 44 tratamos de varrer os diretórios, incluir o arquivo encontrado e na linha 39 é trada a exceção caso o arquivo não seja encontrado. Na linha 47 instanciamos classe da biblioteca, que estará disponível para uso dentro do objeto *libs* do *controller* *($this->libs->bibliotecaEscolhida*) ou *model* (dependendo de onde a biblioteca tenha sido invocada).

Figura 31 – Arquivo que permite carregar uma biblioteca para uso



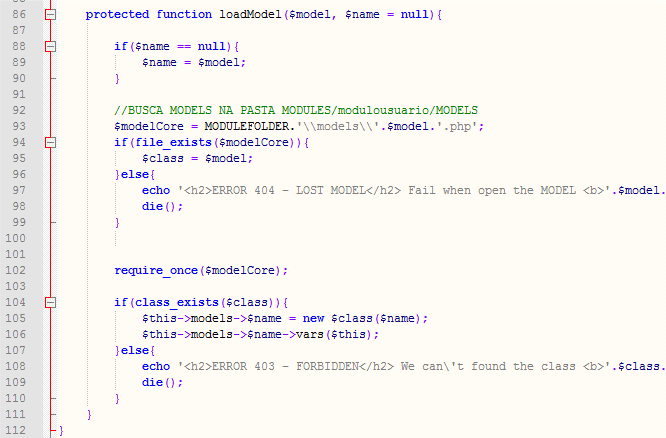
O uso das bibliotecas baseia-se na orientação a objetos, como todo o *framework*. Uma classe é instanciada e somente após é possível utilizar seus métodos e acessar seus atributos.

## A CAMADA MODEL

A camada *model* será responsável por permitir o acesso aos dados do sistema, sejam eles contidos em um banco de dados, arquivos ou coleta de medições do mundo externo. Ele nunca atenderá uma requisição feita pelo *browser* do usuário como visto nas sessões 3.2 a 3.4, e em ambientes de produção não fará exibições na tela como explica a sessão 3.6.2 mas permitirá que o *debug* seja feito em ambientes não produtivos.

Para a camada *model* ser utilizada vamos permitir que ela seja invocada somente de *controllers* criados pelo programador, que ficará disponível um objeto *models* *($this->models->modelEscolhido*). Retornando no controller, vamos criar a função *loadModel()*. Esta função permitirá que o *model* seja utilizado por aquele *controller*, e chamar suas funções quando necessário.

Figura 32 Arquivo função que carrega um *model*



A função *loadModel()* disponibilizará um objeto chamado *models* e que dentro dele ficará todos os models carregados no momento. Podemos ter um *model* que trate de usuários, outro que trate de pedidos de vendas, outro de envio de e-mails, etc, todos disponíveis em um único *controller.* Todos eles podem comunicar-se com o *controller* e com outros *models* apenas.

Como podemos ver na Figura 32 ao chamar a função precisamos passar por parâmetros o nome do model criado previamente dentro do diretório *modules/relatorio/models* e um nome caso faça-se necessário. Na linha 94 validamos se o *model* existe, na linha 102 requeremos que o arquivo seja adicionado a execução do script e por fim na linha 105 instanciamos o *model*.

Para criarmos um model e acessar um banco de dados vamos criar a classe Banco no arquivo *banco.php* no diretório *modules/relatorio/models*. Nesta classe podemos fazer toda a organização de acesso a banco de forma abstraída. Ainda não acessaremos os dados. Ainda teremos mais uma camada entre o *model* e os dados.

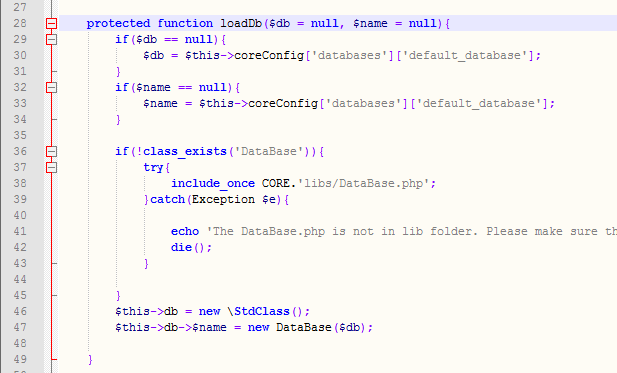
## BILIOTECAS

As bibliotecas nos fornecerão ferramentas para abstrair o uso de terminadas ferramentas como criação de *queries SQL*, criação de paginação, sessões, entra outras. Elas têm o intuito de agilizar e reutilizar código no processo de desenvolvimento entregando facilidades no uso de ferramentas. Algumas dessas bibliotecas fornecem uma abstração bastante forte, como é o caso da biblioteca de manipulação de dados *(*veremos na sessão 3.6.5.1), que isola totalmente o programador da sintaxe de linguagem *SQL* ao conectar com bancos de dados diferentes.

## BIBLIOTECA DE MANIPULAÇÃO DE DADOS

Dentro do *model* e somente dentro dele, poderemos invocar uma classe para manipulação de dados. Podemos acessar bancos de dados, ler e escrever arquivos, capturar dados do mundo real, assim como descreve a premissa de um *model* de *framework* *MVC*. Mas vamos trabalhar apenas com acesso a banco de dados nesta oportunidade.

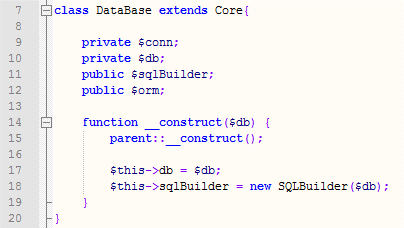
Figura 33 – Função que adapta a classe de conexão ao banco



Como auxílio a camada model, podemos criar uma biblioteca que facilite o acesso a linguagem *SQL*. No arquivo *core/Model.php*, criamos a função *loadDb()*, assim como descreve a Figura 33. Nas linhas de 29 a 34 tratamos qual será o nome do objeto que será disponibilizado no *model* para acessar os dados efetivamente. Na linha 38 incluímos o arquivo *core/libs/DataBase.php* a execução do script.

Tendo em vista que a criação de um *CRUD* [[13]](#footnote-13) é um processo moroso e inclui a manipulação de *SQL* [[14]](#footnote-14) iremos criar uma biblioteca para criar as *queries* de forma semiautomática. No arquivo *core/libs/DataBase.php* criamos a classe *Database que fará* a disponibilidade dos objetos dentro de um objeto no *model ($this->db->bancoEscolhido)*. Esta classe servirá apenas como um conector entre as classes *Model* e *DataBaseConnector* que podemos ver na Figura 35.

Figura 34 – Construção da classe *DataBase*



A classe *DataBaseConnector* tratará a conexão ao banco. Nela vamos tratar as exceções e diferenças ao conectar em bancos de dados diferentes (como *Oracle* e *Postgres*). Esta classe baseia-se no uso do PDO[[15]](#footnote-15) do *PHP* por isso suas respectivas extensões de bancos de dados devem estar ativas nas configurações do *PHP*. Além de fazer a conexão ao banco dados ela entregará a conexão já aberta à classe *DataBase*, e que consequentemente estará disponível no *model* criado pelo programador.

Figura 35 – Classe que trata a conexão ao banco de dados



E por fim na Figura 36 a classe *SQLBuilder* que se encarregará de auxiliar o desenvolvimento de consultas *SQL*, para que o programador não tenha que se preocupar com os diferentes modos e particularidades das *queries* entre diferentes bancos.

Ao utilizarmos esta classe estaremos distanciando o desenvolvedor da linguagem *SQL*. Ainda não será uma abstração completa como um sistema de persistência de dados, mas estamos evitando que ele se preocupe com a sintaxe. Ainda assim ele precisará apenas se preocupar com a ordem das estruturas de uma consulta *SQL* como vemos na Figura 37.

Dessa forma ao surgir a necessidade de trocar o SGBD, não precisamos nos preocupar em refazer todas as *queries* do sistema, bastando apenas inserir os dados corretos de conexão e acesso no arquivo de configuração do sistema. Utilizaremos esta forma de criar queries no *framework*.

Figura 36 – Classe SQLBuilder que permitirá criar consultas *SQL*

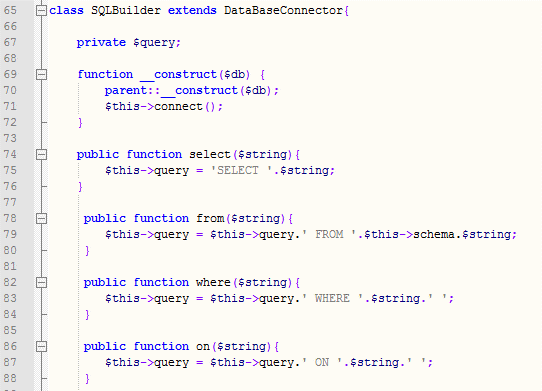
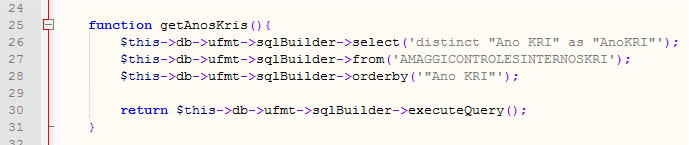


Figura 37 – Criação de um consulta usando o sqlBuilder



Por fim criamos a função *executeQuery()* na classe *SQLBuilder* que fará a execução da consulta no banco, as adaptações pertinentes a este SGBD e retornará um *array* com todos os registros encontrados segundo a *query* criada pela própria classe. Mas podemos criar a *query* manualmente e simplesmente mandar que a classe execute usando a função *setQuery()*. Assim não prendemos o desenvolvedor a usar a classe *SQLBuilder*.

Figura 38 – Função que executa a consulta no banco de dados

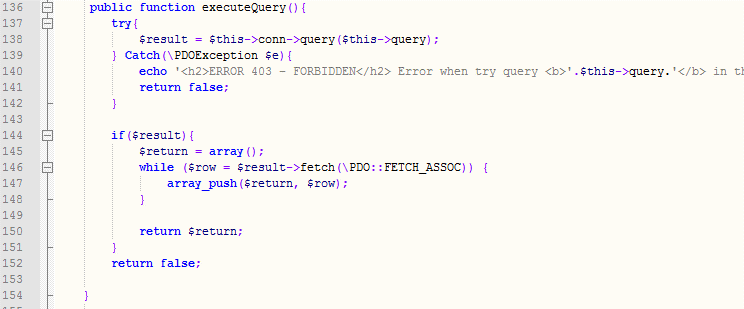


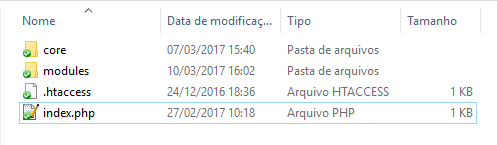
Figura 39 – Função que permite escrever uma consulta manualmente



## ESTADO DO *FRAMEWORK*

Nesta etapa o *framework* está pronto para o uso. Sua estrutura de diretórios ficou como na Figura 40. No diretório *core* temos o núcleo do *framework*, bibliotecas padrões de acesso público, arquivos de aviso de erro, e as classes que criam a estrutura do *MVC*. É interessante que não seja feita alterações no núcleo do *framework* pelos desenvolvedores que farão uso do *framework*.

Figura 40 – Estrutura de diretórios do *framework*

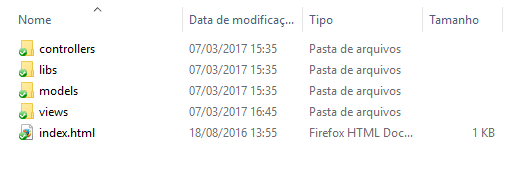


No diretório *modules* deve ser adicionado um diretório para cada módulo do sistema e podemos ter tantos módulos quanto necessário. É possível ter um módulo para cada assunto do sistema como contabilidade, estoque, gerencia de usuários, entre outros. O módulo é o primeiro parâmetro que passamos via *URL*. E ainda é aqui que adicionamos o arquivo de configuração.

Na Figura 41 vemos a estrutura de um módulo. Cada um dos módulos deve conter esta mesma estrutura conforme a filosofia do *MVC*. Ainda podemos adicionar bibliotecas privadas na pasta *libs*, tendo em vista que estas só estarão disponíveis para este módulo apenas. Nela podemos adicionar bibliotecas de terceiros ou criar uma biblioteca própria.

Caso seja necessário fazer um compartilhamento entre bibliotecas e classes entre módulo, pode-se criar o módulo *default* e nele adicionar os arquivos necessários. Sendo assim, em qualquer *controller* ou *model* de qualquer módulo, basta chamar utilizando a funções *loadLib()* ou criando uma instancia comum da orientação a objetos*.*

Figura 41 – Estrutura de um módulo



## APLICAÇÃO E USABILIDADE

A proposta da utilização de um *framework* *MVC* é agilizar, simplificar o processo de desenvolvimento e dividir o código em interesses, deixando-o mais limpo e organizado, além de trazer algumas ferramentas e bibliotecas para facilitar a vida do desenvolvedor. Para isso fragmentamos um código que poderia estar em um único arquivo *PHP* em vários arquivos organizados, facilitando a identificação de erros e bugs. Isso aumenta a quantidade de arquivos de um sistema, mas em contrapartida, permite a reutilização de código, bibliotecas, classes, páginas e outras dependências.

Iniciaremos então uma breve comparação com uma página de relatório da auditoria e o tratamento de não conformidades da empresa Amaggi sem o uso do *framework* e com o uso do *framework*, explicitando as diferentes sessões de uso de cada do código.

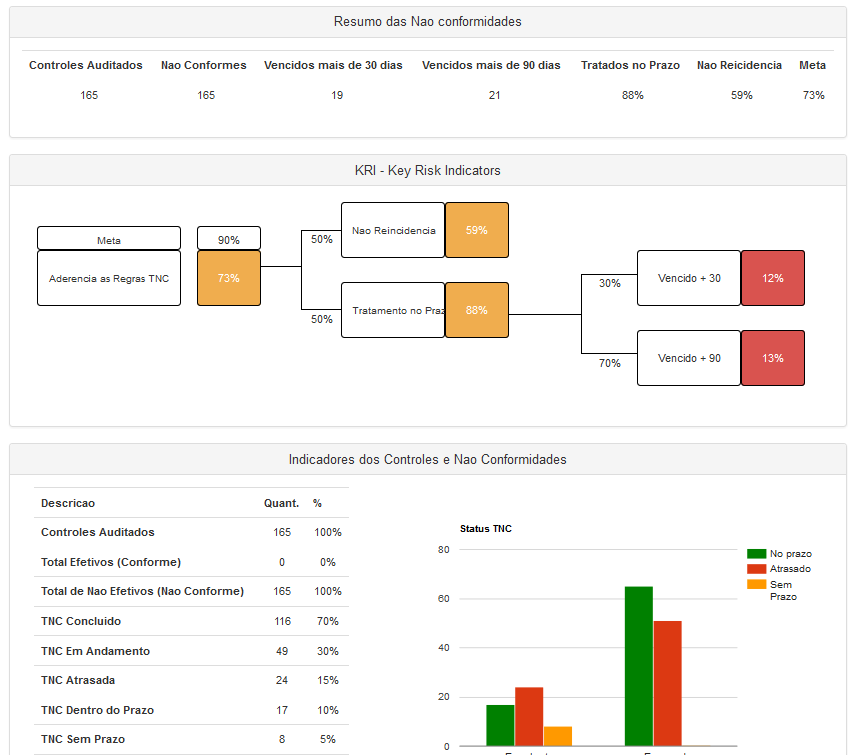
## O RELATÓRIO DE NÃO CONFORMIDADES EM PLANILHAS

Inicialmente o relatório de não conformidades e o tratamento de não conformidades apontadas pela auditoria (*KRI*[[16]](#footnote-16)) era feita de forma manual, tanto a alimentação dos dados quando atualizados, registros de controles, cálculos de metas da empresa e estatísticas básicas eram feitas em planilhas. Eram utilizadas uma planilha para cada filial auditada para cada ano, isso nos dá aproximadamente 30 planilhas anuais. Este quadro foi revertido com a automação do processo de tratamento de não conformidade (*TNC*) com o uso do SoftExpert. Todos os dados são guardados em um banco de dados Oracle, e podem ser facilmente extraídos via *SQL*.

## O RELATÓRIO DE NÃO CONFORMIDADES INTERFACE *WEB*

Foi criado uma página *web* que faz a extração dos dados necessários e apresenta o relatório e alguns indicadores necessário para que o *KRI* seja gerado. Toda a alimentação de dados é feita pelo BPMS. Todas as necessidades da página como conexão com o banco, cálculos estatísticos e o *HTML* estão dentro de um único arquivo *PHP* (exceto dependências *HTML* como bibliotecas de estilo e *javascript*). O arquivo ficou com um tamanho total de 33KB, contendo 883 linhas. Sem a implementação de orientação a objetos o arquivo ficou extenso e complexo de entender. Não há divisão explicita do código e nem blocos concretos que dividem os interesses.

Figura 42 – Pagina *web* do relatório de não conformidades



Essa apresentação *web* de um sistema que anteriormente era controlado via planilha, já trouxe uma melhoria significativa para o monitoramento e gerencia do *KRI*. Mas outros problemas foram gerados, como por exemplo o tempo de resposta da consulta que gera o *KRI* é de 40 a 50 segundos, um tempo bastante alto quando se precisa fazer 30 filtros para que as informações sejam coletadas para montagem de apresentações para a diretoria da empresa. O único SGBD que este único arquivo consegue conectar é o próprio Oracle, e não há uma maneira simplificada de implementar um sistema de cache. Por conta destes problemas iniciamos a implementação do *KRI* no *framework* desenvolvido.

## APLICAÇÃO DO *FRAMEWORK* NO *KRI*

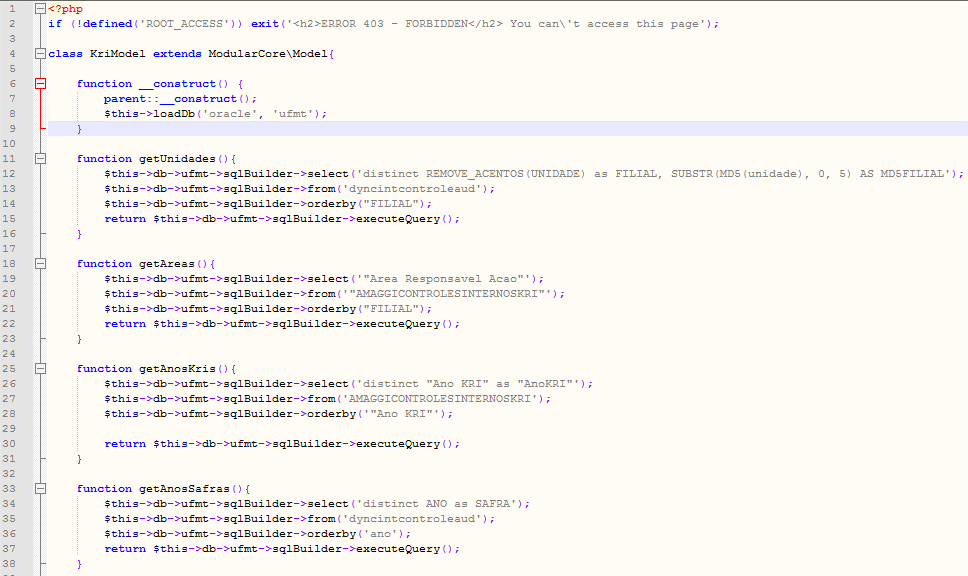
Basicamente o relatório do *KRI* coleta os dados dos controles geridos pela auditoria e confronta com os dados dos *TNC* registrados no sistema. É necessário coletar alguns dados do banco de dados antes de cruzar estes dados. Depois dos dados coletados, usaremos estas informações dentro de um *SQL* que nos trata todos os dados do *KRI*. Após precisamos fazer alguns cálculos para obter os valores a serem apresentados nos gráficos.

## MODEL

No *model*, faremos a conexão com o banco de dados Oracle e faremos algumas *queries* previamente necessárias. Como já temos todos os dados que precisaremos, basta implementarmos as funções que serão utilizadas posteriormente pelo *controller*. Vamos utilizar a classe *sqlBuilder* nativa do model para criarmos as *queries*, assim teremos a flexibilidade de podermos mudar de SGBD sem esforço nenhum, uma vez que toda sintaxe da consulta é flexível ao funcionamento do sqlBuilder.

Basicamente não é necessário digitar nenhuma linha de *SQL*, mas apenas fragmentos como mostrado na Figura 43. O *sqlBuilder* tratará de gerar a consulta, fazer uma adaptação dos dados retornados do banco para um *array* puro, contendo como chave o nome da coluna da tabela e como valor o valor da coluna para cada registro retornado.

Figura 43 – Consultas de coleta de dados para gerar o *KRI*



## CONTROLLER

Já temos todos os dados retornáveis do banco de dados. No *controller* então faremos apenas a chamada dessas informações e sua gerência. Nas linhas 51 a 54 da Figura 44, fazemos a chamada das funções do model que registrará em variáveis do tipo *array* que utilizaremos para adiante. Note que os valores são retornados para variáveis soltas e para o *array* $*coreView*. O motivo disso é que além de usarmos os dados retornados para realizar alguns cálculos, precisaremos deles para apresentar ao usuário.

Figura 44 – Chamada das funções do model retornando os dados do *KRI*

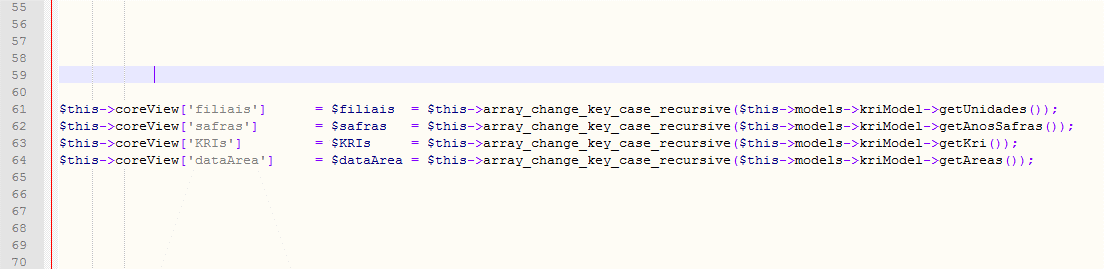
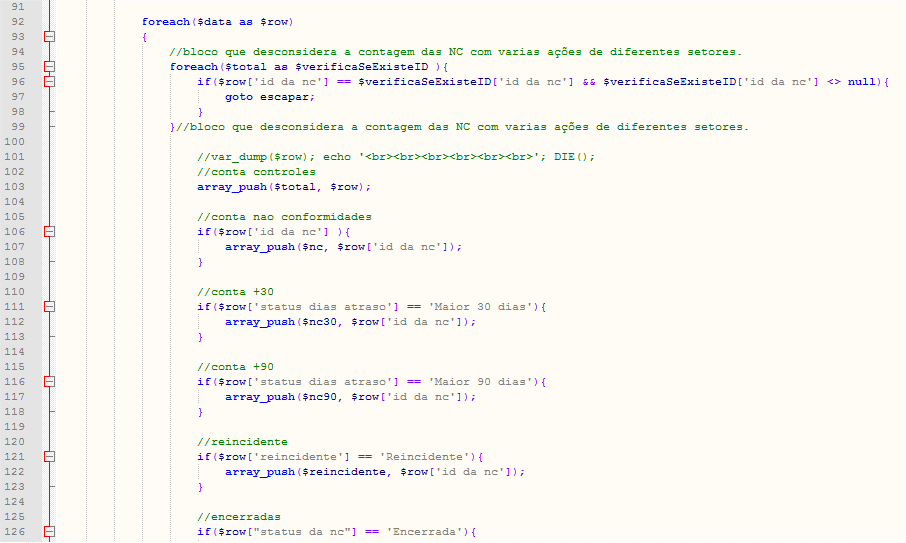


Figura 45 – Classificação dos dados extraídos



Feito a coleta de dados partiremos para a classificação dos dados extraídos. Esta é uma etapa que precisaremos varrer cada registro e verificar alguns dados, e segundo valores alimentaremos algumas variáveis. Por fim realizamos os cálculos estatísticos e guardamos os dados unicamente no *array* $*coreView* para apresentarmos na tela ao usuário como mostra a Figura 46.

Figura 46 – Realização dos cálculos estatísticos



## VIEW

Temos todos os dados coletados e processados, na view nos resta apresentar os dados. A *view* é composta em sua maioria por códigos *HTML*. Apenas em locais estratégicos são apresentados dados gerados no *controller*. Os únicos trechos de código *PHP* que a teoria *MVC* permite é o de estruturas de repetição da linguagem. Em resumo a *view* ficou com um código extremamente simples.

Figura 47 – A apresentação em *HTML* dos valores coletados no Controller

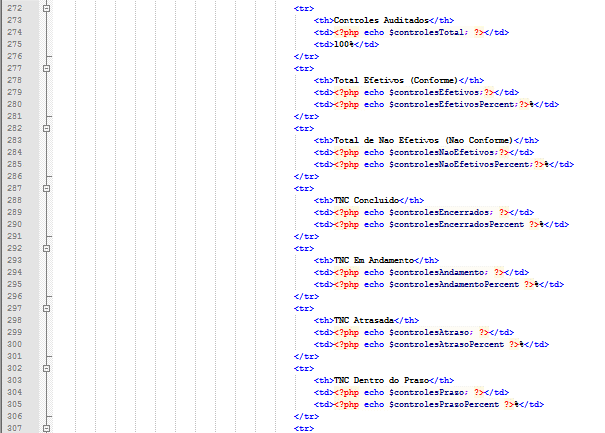


Figura 48 – Estrutura de repetição dentro da view



## RESUMO DA APLICAÇÃO DO *FRAMEWORK*

Ao realizar a migração do relatório, podemos analisar algumas características. O número de arquivos aumentou, mesmo porque separamos um único código em 3 arquivos distintos. Quanto ao número de linhas o código sem o uso do *framework* dá um total de 883 linhas sem definição clara de blocos de código. Usando o *framework*, o *controller*, *model* e *view* ficaram com respectivamente 284, 205 e 521 totalizando 1010 linhas, um aumento de 14%. Em contrapartida temos uma separação de interesses bem definida.

Também foi realizado uma medição de tempo em que as páginas são geradas pela função *microtime()* [[17]](#footnote-17)do *PHP*. Sem o uso do *framework* obtivemos a geração da página completa inclusive com o tempo de consulta no banco 3.101s. Realizamos o mesmo com o uso do *framework* e obtivemos 3.150s. Consideramos este tempo razoável tenho em vista toda a complexidade das classes do *framework*.

# DIFICULDADES ENCONTRADAS

Uma das principais dificuldades encontradas na execução desse trabalho foi o fato de todas as etapas: análise, desenvolvimento, arquitetura e publicação do *framework* e sua aplicação no relatório do *KRI*, ferramenta relativamente complexa, foram realizadas por uma única pessoa.

Outra dificuldade foi dada o tempo que se teve para executar as tarefas do estágio e redigir o relatório, definir quais as funcionalidades da ferramenta seriam priorizadas e implementadas, e ainda, adquirir alguns conhecimentos necessários para o desenvolvimento dessas funcionalidades. Nem todas as bibliotecas que achamos interessantes foram implementadas.

A falta de tempo é o maior inimigo para a criação de um sistema. Por isso um dos objetivos do *framework* é exatamente deixar que o desenvolvedor não se preocupe com funções básicas da linguagem, e se dedique mais às regras de negócios. Esse déficit de tempo inclusive atrapalhou no desenvolvimento do *framework* e dada essa situação, priorizou-se a entrega de uma ferramenta que prezasse a entrega de uma ferramenta que atendesse o mínimo do *KRI* e conseguisse buscar os propósitos de criação do *framework*.

Outro ponto de dificuldade foi que pouco se conhecia sobre o SoftExpert (BPMS) e o banco de dados Oracle. Foi dedicado um período de tempo para o estudo funcionamento do SoftExpert que é bastante amplo, com vários módulos, regras de negócios e particulares, e ainda um tempo para o estudo da sintaxe das consultas do Oracle. Mesmo que o Oracle e o PostgreSQL sejam bancos de dados relacionais com a padronização das consultas *SQL*, as estruturas delas são iguais, mas suas sintaxes são diferentes e com muitas particularidades.

# CONCLUSÕES

O intuito do projeto era criar um *framework* *PHP* que agilizasse o desenvolvimento de páginas *web* removendo tarefas rotineiras como analisar e alterar a sintaxe de conexão ao banco de dados, e a sintaxe consultas na migração de um banco para outro. Usando o *framework* apenas passamos qual banco queremos acessar e as credenciais. O *framework* utiliza a biblioteca correta e se encarrega de gerar a sintaxe.

Outro propósito era a separação explicita de código por interesses. Existe um bloco especifico que trabalha com acesso aos dados, um especifico que organiza os dados coletados e um último que apresenta os resultados ao usuário. A estrutura de classes que cria esta separação ainda permitiu que essas divisões por interesse conversem entre si de forma organizada e padronizada, e ainda que acessemos outras classes criadas pelo desenvolvedor e bibliotecas de forma simplificada.

O *framework* carrega os dados de configuração e transporta para cada classe e biblioteca, dispensando a tarefa do desenvolvedor de carregar estas configurações em cada parte do seu sistema. Se um novo parâmetro de configuração for necessário na aplicação, basta adicioná-lo no arquivo de configurações que ele será distribuído.

O *framework* foi criado de forma flexível e aceita bibliotecas de terceiros que podem ser importadas usando um gerenciador de dependências, ou o desenvolvedor pode criar suas próprias bibliotecas e classes. Pode se considerar que o *framework* encontra-se em uma versão inicial, e como implementações futuras está previsto a criação de bibliotecas para sistema de login com uso de sessões e segurança (permissões de acesso), encriptação, paginação, e-mail, estáticas e geração de gráficos entre outras.

# REFERÊNCIAS BIBLIOGRÁFICAS

ANTUNES, Jonathan Lamim. **CODEIGNITER – Produtividade na criação de aplicações web em PHP**, 1ª ed. São Paulo: Casa do Código, 2016.

KABIR, Mohammed J., 2002. **Apache Server 2 Bible**,1ª ed. EUA: Hungry Minds, Inc

PITT, Chris; 2012**. PRO PHP MVC – Everything you need to know about using MVC with PHP in a single reference**, 1ª ed. EUA: Springer Science+Business Media New York.

POSTGRESQL, The PostgreSQL Global Development; 2017. **PostgreSQL 9.5.6 Documentation**. Disponível em <https://www.postgresql.org/files/documentation/pdf/9.5/postgresql-9.5-A4.pdf>. Acesso em 27/02/2017

OGLIO, Pablo Dall’; 2016. **PHP – Programando com Orientação a Objetos**, 3ª ed. São Paulo: Novatec Editora

MATTSSON, Michael; 2000. **Evolution and Composition of Object-Oriented Frameworks**, EUA: University of Karlskrona/Ronneby

OLIVEIRA, Rommel. **BPM, BPMS e BPMN, quais as diferenças?**. Disponível em: <http://www.lecom.com.br/blog/2015/11/05/bpm-bpms-e-bpmn-quais-as-diferencas/>. Acesso em: 27/02/2017

1. C# - É uma linguagem de programação projetada para criar uma variedade de aplicativos executados no .NET Framework. <https://msdn.microsoft.com/pt-BR/library/kx37x362.aspx>, acessado em 13/03/2017 [↑](#footnote-ref-1)
2. Debug – É a busca e correção de erros e problemas no processo de desenvolvimento de softwares. <http://searchsoftwarequality.techtarget.com/definition/debugging> [↑](#footnote-ref-2)
3. Git – Sistema de versionamento de software. <https://git-scm.com/> [↑](#footnote-ref-3)
4. .htaccess – Arquivo que oferece um meio de fazer mudanças nas configurações do Apache por-diretório. <https://httpd.apache.org/docs/2.2/pt-br/howto/htaccess.html> acessado em 13/03/2017 [↑](#footnote-ref-4)
5. $\_GET - Um array associativo de variáveis passadas para o script atual via o método HTTP GET. <http://php.net/manual/pt_BR/reserved.variables.get.php>, acessado em 13/03/2017 [↑](#footnote-ref-5)
6. require\_once() – Função global do PHP que permite agregar um arquivo a execução atual do script. <http://php.net/manual/pt_BR/function.require-once.php>, acessado em 13/03/2017 [↑](#footnote-ref-6)
7. die() – Função global do PHP que mata a execução do script bruscamente. <http://php.net/manual/pt_BR/function.die.php>, 13/03/2017 [↑](#footnote-ref-7)
8. \_\_construct() – Função construtora das classes PHP. <http://php.net/manual/pt_BR/language.oop5.decon.php>, acessado em 13/03/2017 [↑](#footnote-ref-8)
9. Echo() – Função da linguagem PHP que exibe uma ou mais strings. Disponível em <https://secure.php.net/manual/pt_BR/function.echo.php>, acessado em 09/01/2017 [↑](#footnote-ref-9)
10. ob\_clean() – Função da linguagem PHP que apaga o buffer de saída dos dados que serão enviados a tela do usuário. Disponível em: <http://php.net/manual/pt_BR/function.ob-clean.php>, acessado em 09/01/2017 . [↑](#footnote-ref-10)
11. var\_dump() - Função da linguagem PHP que explora uma determinada variável ou objeto e destrincha todo o seu conteúdo exibindo-o na tela. [↑](#footnote-ref-11)
12. XDebug – Extensão do PHP que facilita o debug do código. Disponível em <https://xdebug.org/>, acessado em 10/01/2017. [↑](#footnote-ref-12)
13. CRUD – CRUD vem do inglês (Create, Read, Update, Delete) que quer dizer basicamente as quatro principais operações com um banco de dados (inserir, ler, atualizar, excluir). <http://www.devmedia.com.br/crud-com-php-pdo/28873>, acessado em 24/01/2017. [↑](#footnote-ref-13)
14. SQL - (Structure Query Language - Linguagem de Consulta Estruturada) é a linguagem padrão ANSI (American National Standards Institute - Instituto Nacional de Padronização Americano) para a operação em **bancos de dados** relacionais. <http://www.luis.blog.br/o-que-e-sql.aspx>, acessado em 24/01/2017. [↑](#footnote-ref-14)
15. PDO – É uma extensão do PHP que permite criar uma conexão com vários bandos de dados de diferentes fabricantes. <http://php.net/manual/pt_BR/intro.pdo.php>, acessado em 13/03/2017 [↑](#footnote-ref-15)
16. KRI - Key Risk Indicators ou Principais Indicadores de Risco - são componentes do processo de monitoramento de riscos de uma empresa que são utilizados para fornecer indicadores. <http://www.ibm.com/support/knowledgecenter/pt-br/SSFUEU_7.2.0/com.ibm.swg.ba.cognos.op_app_help.7.2.0.doc/op_app_help_plugin-gentopic66.html>, acessado em 27/02/2017 [↑](#footnote-ref-16)
17. Microtime() – Função PHP que retorna um timestamp unix. <http://php.net/manual/pt_BR/function.microtime.php>, acessada em 27/02/2017 [↑](#footnote-ref-17)